AD-A056 771 MITRE CORP BEDFORD MASS F/G 9/2
SOFTWARE DESIGN METHODOLOGIES AND AIR FORCE SOFTWARE ACQUISITIO--ETC(U)
JUN 78 D L JAMES F19628-77-C-0001
UNCLASSIFIED MTR-3508 ESD-TR-78-147 NL

1 OF 2
ADA
056771

LEVEL II

(18) ESD-TR-78-147 (19)

(12) (14) MTR-3508

(6)

SOFTWARE DESIGN METHODOLOGIES AND AIR FORCE SOFTWARE ACQUISITION ENVIRONMENT.

(10) D. L. JAMES

(11) JUN 78

(12) 127p.

Prepared for

ROME AIR DEVELOPMENT CENTER
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Massachusetts

(9) Technical rept.

Project No. 522M

Prepared by

THE MITRE CORPORATION
Bedford, Massachusetts

Contract No. (15) F19628-77-C-0001

235 050

When U.S. Government drawings, specifications, or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise, as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

Do not return this copy. Retain or destroy.

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

WILLIAM E. RZEPKA
Project Engineer

DEANE F. BERGSTROM
Ch, Info Mgt Sciences Section

WENDALL C. BAUMAN, COL, USAF
Ch, Info Sciences Division

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-78-147	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SOFTWARE DESIGN METHODOLOGIES AND AIR FORCE SOFTWARE ACQUISITION ENVIRONMENT		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER MTR-3508
7. AUTHOR(s) D. L. James		8. CONTRACT OR GRANT NUMBER(s) F19628-77-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS The MITRE Corporation P. O. Box 208 Bedford, MA 01730		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Project No. 522M
11. CONTROLLING OFFICE NAME AND ADDRESS Information Processing Branch Rome Air Development Center Griffiss Air Force Base, NY 13441		12. REPORT DATE JUNE 1978 13. NUMBER OF PAGES 125
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Electronic Systems Division, AFSC Hanscom Air Force Base, MA 01731		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

HIERARCHICAL DESIGN METHODOLOGIES (SRI)
RATIONAL DESIGN METHODOLOGY (RADC/Honeywell)
SOFTWARE ACQUISITION
SOFTWARE DESIGN METHODOLOGIES

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report represents the final report for the Design Methodology Transfer subtask of the Formal Design Methodologies Task of Project 5220. The general goal of the subtask was to explore the ease of transfer of technology on software design methodologies to the Air Force software acquisition environment by studying two methodologies selected by RADC, comparing them with the acquisition environment, and discussing them with

Power

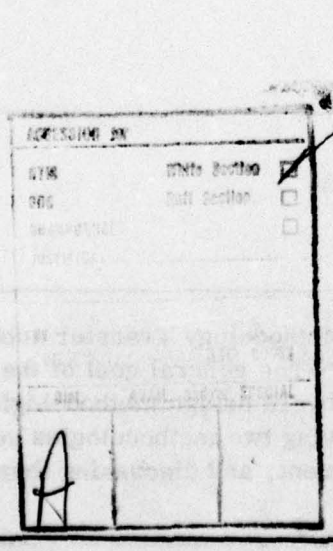
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (concluded)

acquisition personnel. The two methodologies (one by SRI and one by Honeywell, Inc.) are described with examples. The assumed acquisition environment is described and compared with the methodologies. The presentations to acquisition personnel are described. Their views as well as those of Project 522M personnel on application of the methodologies are discussed.



ACKNOWLEDGMENTS

This report has been prepared by The MITRE Corporation under Project 522M. The contract is sponsored by the Rome Air Development Center, Griffis Air Force Base, New York, and the Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Massachusetts.

The author of this report acknowledges and appreciates the efforts of a number of people who contributed to the work described herein:

Jon Millen of MITRE D-75 for preparation for this project of the example of the SRI methodology in Section III and for discussions of the methodologies with the author.

Don Boyd and associates at Honeywell, Inc., for discussing and answering many questions about the Honeywell methodology.

Karl Levitt and associates at SRI International for discussing and answering many questions about the SRI methodology.

Harlan Mills, Andy Ferrentino, Max Wilson and associates at IBM for several discussions of their work in software design methodologies.

Personnel from ESD/MCV and MITRE D-74 for arranging and participating in the SATIN IV discussions of the methodologies.

Personnel from ESD/DCO and MITRE D-71 for arranging and participating in the TACC Automation discussions of the methodologies.

John Glore of MITRE D-75 for a number of discussions of the Air Force software acquisition environment.

Bill Rzepka of RADC, who monitored the work described in this report, for a variety of contributions to the work.

Frazier Patrick of MITRE D-81 and Judy Shapiro of MITRE D-61 for discussions of the Air Force software acquisition environment.

Pete Veckery of ESD/MCI for discussion of the Air Force software acquisition environment and software design methodologies.

Marlene Hazle of MITRE D-75 for many discussions of software design methodologies, the Air Force software acquisition environment, and this report.

Jane McCarthy of MITRE D-75 for her considerable effort and care in typing this report and preparing most of its figures.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

LIST OF ILLUSTRATIONS AND TABLES

SECTION I

INTRODUCTION

This section discusses (1) the work performed during FY77 on the Design Methodology Transfer subtask of the Formal Design Methodologies task of Project 5220 and (2) the contents of the other sections of this report. The subtask was performed in support of RADC's software design methodology work.

ACTIVITIES

The general goals of the subtask were to study software design methodologies, to consider the application of two specific ones to the Air Force software acquisition environment, to hold discussions with experienced acquisition personnel from ESD and MITRE about the methodologies and their application to the Air Force environment, and to report the opinions of acquisition personnel and personnel from this project as to the advantages and disadvantages of the methodologies and desirable changes to the methodologies and/or the acquisition environment to allow their utilization. The general emphasis was on steps RADC might take to foster the transfer of technology on software design methodologies, primarily from the private sector, to the Air Force software acquisition environment.

A short review of literature in the field of design methodologies was conducted during the beginning of the contract year based in part on citations and papers gathered during the previous contract period by other people. Based on this review and inputs from RADC, a set of six methodologies was selected for more detailed study. The remainder of the first half of the contract year was largely devoted to study of the six methodologies and to characterizing and classifying the six methodologies for an interim report, which was published at the end of this period as a MITRE Working Paper.

It was found during this period that a number of the six methodologies had not been documented very extensively, a particular problem with regard to methodologies to be discussed with acquisition personnel. Two of the six methodologies had been selected by RADC early in the contract period for potential application in the Air Force software acquisition environment. However, based on the evolving nature of software design methodologies and their relative lack of documentation, another of

the six was substituted for one of the two selected ones. The two methodologies finally compared with the acquisition environment were later and different versions (one considerably so) of those studied during the first six months of the contract year. The two methodologies compared with the acquisition environment were the Hierarchical Design Methodology (HDM) of the Stanford Research Institute (now SRI International) and a methodology being developed as of March 1977 by Honeywell, Inc. for RADC, which has been called the Rational Design Methodology (RDM).

A week was spent in early May in discussions with Honeywell and SRI personnel on their methodologies. Various telephone conversations have been held since, primarily with Honeywell, because of the very limited documentation and examples for the RDM as yet. The HDM has been documented in a limited fashion, largely as a by-product of operating system documentation. Based on the discussions and available documentation, Sections III and IV of this report were written to serve as a basis for discussions with acquisition personnel. It is hoped that they represent relatively accurate descriptions of the intentions and views of the developers of the methodologies.

The model of the Air Force software acquisition environment was developed based on study of various regulations, standards, and MITRE reports, a number of conversations with MITRE personnel, and some limited experience in the environment.

REPORT CONTENT

Section II of this report discusses characteristics of software design methodologies, based largely on the six basic ones studied as well as variations of some of these and others reviewed briefly. The characteristics are intended to define the subject matter of this report and the subtask rather than the essential attributes of any software design methodology. Sections III and IV describe the HDM and the RDM, respectively, with examples.

Section V presents and describes the assumed model acquisition environment. Section VI describes the author's views regarding the possible use of the HDM and of the RDM in the model acquisition environment. Areas of possible change in the methodologies or in the acquisition environment are discussed.

Section VII discusses the material presented to acquisition personnel in discussions held with them at the end of August 1977. Section VIII presents the views of the acquisition personnel

concerning the methodologies as expressed in the discussions. Section IX discusses primarily the author's views on the methodologies.

SECTION II

CHARACTERISTICS OF A SOFTWARE DESIGN METHODOLOGY

Various efforts over the last 10 to 15 years have been aimed at making computer software development more systematic, more of an engineering discipline. Although the efforts earlier in this period tended to relate to implementation, various contributions to improved design were also made in the form of codification of good practices and the proposal and/or development of new concepts. Particularly in the last few years, these earlier practices and concepts have been combined in various ways with recent concepts to form more complete methodologies to be used in software development including the design phase.

Since the development of software methodologies, as such, is a rather recent phenomenon, public documentation of a number of them tends to be somewhat rudimentary. Other reasons for the limited documentation are that some of the methodologies are still evolving, some need further development in some areas, and some are treated as proprietary. For these reasons, a number of the methodologies have had limited application, as yet, by a limited number of organizations. However, in toto, a relatively large number of individuals and organizations have done work in the field, and there are some common characteristics in these efforts.

Much of the work accepts as a principle the application of increased effort in the design phase with higher costs, to be more than offset by savings in later phases of the life cycle. The savings would result from: (1) fewer design changes due to errors found during implementation or testing which would otherwise cause iteration of the development steps for at least a portion of the system and (2) less effort to make design changes. Reduced effort for design changes is a principal goal of many methodologies. This reduction affects not only changes due to errors but later software maintenance changes during the operational phase.

A number of the design methodologies are intended to be useful during other development phases as well as design; implementation is frequently included. A much smaller number are related to requirements analysis -- in fact, many methodologies seem almost to ignore the quality and form of the basic requirements information which form their input.

Many methodologies rely upon computers to store the design as it evolves, except for possibly those which use graphic techniques such as HIPO (Hierarchy plus Input-Process-Output) charts and SofTech's SADT (Structured Analysis and Design Technique). Even these may be capable of being stored in a computer in the form of text blocks, symbols to identify the type of block, and linkages between blocks. Storage of the evolving design in a computer generally means much readier and more frequent accessibility to the design by the designers and possibly design monitors. Such design documentation is probably more likely to be up-to-date and available when needed than if treated as a separate effort.

Various methodologies call for the creation of formal design specifications which are more mathematical in nature and more precise than English text. Proofs of correctness at the design and at the implementation levels are facilitated by such specifications although such proofs may not be considered necessary or cost-effective for non-critical software. Storage of such specifications in a computer makes it possible to use the computer to make various kinds of checks such as for consistency.

Many methodologies incorporate program design languages (PDL's) for expressing the design. A PDL which is closely associated with a methodology can reinforce its goals and restrain the designer from following practices considered undesirable. The computer can be used to perform the reinforcement and the restraint.

Tools to support a methodology vary considerably from one methodology to another. A PDL may be considered one type of tool. Automated tools to check the consistency of a design expressed in a PDL and to enforce syntax rules exist. Text editors and data management systems are used for storage and retrieval of designs. Graphic aids are useful to portray the structure of a design. Tools to support proof-of-correctness exist in some cases and represent probably the most complex type of tool and, therefore, the most helpful if proofs are needed.

The final area to be discussed is that of guidelines for use of a methodology. One type of guideline offered by some methodologies is that of considering software development as a series of steps or stages. Each stage should be characterized in terms of the activities performed, types of decisions to be made, and the products and results to be produced.

Another type of guideline consists of recommendations for handling the complexities of a large software system. The usual guideline for handling complexity relates to modularization of the

design, and hierarchical modularization is frequently suggested. Modules generally should be as independent and self-contained as possible to facilitate design changes. To reduce complexity and improve understandability, modules should also be at least relatively small. Such modules should probably aid in providing traceability back to individual requirements.

A basis for modularization supported by both the methodologies discussed in this paper as well as at least several others is that of the abstract data type, which provides for centralized responsibility within one module for all data objects of a given type. That module contains detailed knowledge of the structure of such objects and contains the only programs in the system for manipulating such objects. Other modules in the system (at higher levels) have only a very limited (abstract) view of such objects - knowledge of the kinds of data obtainable from such objects and the operations available upon them. The programs of the responsible module perform the operations, and that module delivers the data requested. A relatively complex, abstract data type provided by one module may be implemented in terms of one or more simpler, abstract data types provided by other modules.

No attempt has been made above to list the characteristics that a software design methodology ought to have. Clearly some of the general goals of such methodologies are to produce better, more reliable software, to reduce the time required to produce software and also the associated costs. The general frame of reference is that of large, complex software systems and not just small, simple ones. The methodologies attempt to make software design a more systematic process than in the past. Designs should be easier to understand. Complexity is reduced in many cases by providing more structure for a design. Tools, conceptual or otherwise, should be suggested as well as rules for their use to achieve desired goals. The use of computers as much as possible to design software for computers seems quite reasonable.

SECTION III

STANFORD RESEARCH INSTITUTE METHODOLOGY

Development of software design methodologies at the Stanford Research Institute (SRI) has been underway for at least four or five years. References in the literature to what is now known as the Hierarchical Design Methodology (HDM) appeared as early as 1974. The HDM is described as a formalization of the stepwise refinement concept of Dijkstra (Dijk72). The abstract machines of the HDM are represented as Parnas modules (Parn72a, Parn72b).

The principal developers of the HDM are P. G. Neumann, K. N. Levitt, and L. Robinson, assisted by other people at various times. Documentation of the HDM as a whole has suffered by virtue of its treatment as one component of documentation of its largest application. Thus, the most extensive documentation of the HDM appears in several sections of Neum75. An extended and updated version (Neum77) of that report describes more briefly a somewhat modified form of the HDM. Several of the components of the HDM have recently been documented (Robi77, Roub77, Boye76) separately from their application. The basis for this report includes the 1976 and 1977 reports mentioned; personal communications with SRI personnel, primarily K. N. Levitt; and, more informally, earlier SRI papers, including Neum75, which had been reviewed previously.

The largest application of the HDM to date is the design of a Provably Secure Operating System (PSOS), the principal subject discussed in Neum77, Neum75, and earlier papers. The PSOS development has been carried out for the first three of the five stages of the HDM. The HDM has been and is being applied by SRI to other, smaller projects, whose results were not reviewed for this report. The HDM is said to be in use at the University of Texas; a similar methodology was developed concurrently at The MITRE Corporation and applied in computer security work. The HDM has also been examined in detail by at least one branch of a government agency and at least one commercial organization, presumably with regard to its possible application by them.

ABSTRACT MACHINES - BASIS FOR METHODOLOGY

A Single Abstract Machine

The basic concepts on which the SRI Hierarchical Design Methodology (HDM) is based appear to be the following. As noted by others, what SRI refers to as an abstract machine is a finite-state machine (FSM). Assume for the moment that an FSM is a mechanism which is composed of a collection of hardware and/or software whose exact nature we are not concerned with now. This mechanism is capable of performing a defined set of operations, each of which produces a defined effect on the state of the mechanism. Given that the mechanism is in state sk at time tk and that operation oj is then performed, at the completion of the operation (at time tk+1) the mechanism will be in state sk+1. The final state, sk+1, is clearly dependent on the initial state, sk, and the operation performed, oj. The state of the mechanism during the execution of an operation is undefined. We are now concerned only with its state before and after the execution of each operation, and at this point we can only make statements about its state at such points in time. (No assumptions are made as to the amount of time the mechanism requires to execute each operation; tk and tk+1 are simply the points in time prior to the execution of an operation and after execution of it, respectively.)

SRI defines the state of such a mechanism (at the points in time at which state is definable) in terms of a set of state variables, V1, ..., Vn. A given state can be characterized by an n-tuple containing the value of each state variable, (V1, V2, ..., Vn). The state space of the mechanism consists of the Cartesian product of the state variables, V1 x V2 x ... x Vn. (The number of possible states for the mechanism is the product of the number of possible values for each state variable.)

A specification for such a mechanism (FSM or abstract machine) defines the mechanism in terms of the operations it can perform and the operations in terms of the state changes they produce. As SRI states, such specifications define everything a user of the mechanism needs to know about it and everything he is allowed to know, for purposes of the HDM. A specification for such a mechanism defines its state variables and makes assertions as to the effect of its operations on its state. The user of such a mechanism may only change the state of the mechanism by requesting it to perform the defined operations (one at a time) and may determine the state of the mechanism (between operations) by requesting the current values of the state variables.

Hierarchy of Two Abstract Machines

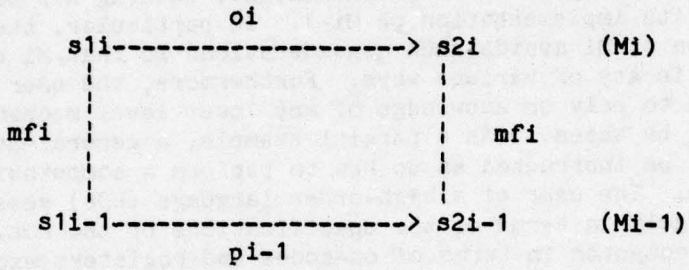
The scope of the discussion is now expanded to include two such mechanisms or abstract machines. Each is defined by a separate specification of the type discussed above. The purpose of discussing two abstract machines is that we wish to consider implementing one abstract machine, Mi, on (in terms of) a second abstract machine, Mi-1. Thus, a two-level hierarchy of abstract machines is created with Mi being at the higher level. In defining (specifying) the higher-level mechanism, Mi, nothing has been said to preclude its implementation on Mi-1. In particular, the specification of Mi avoids such considerations so that Mi can be implemented in any of various ways. Furthermore, the user of Mi is not intended to rely on knowledge of any lower level mechanism on which it may be based. (As a partial example, a general-purpose computer can be instructed as to how to perform a computation in several ways. The user of a high-order language (HOL) sees the computer largely in terms of the specifications of the HOL. He does not see the computer in terms of op-codes and registers except at times when efficiency must be considered. Such features of the computer are hidden from the user by the HOL. Furthermore, he is largely not concerned with how his program is executed, whether his HOL code is translated directly to machine code or whether assembler code is produced as an intermediate form, or even whether a conventional computer is used at all. This statement ignores possible efficiency and cost considerations and assumes that the environment offered to the user is not only adequate for him to describe his problem but also for him to debug his program without resorting to assembly listings or machine dumps.)

If the upper-level abstract machine, Mi, is to be implemented in terms of Mi-1, a method of relating the two machines is necessary. First, a mapping (called a mapping function by SRI), mfi, is needed to relate the state of Mi to that of Mi-1. If Mi is to be implemented in terms of Mi-1, then the state space of Mi-1 must be at least as large (contain at least as many unique states) as that of Mi and each state in Mi must correspond to at least one unique state in Mi-1. Otherwise, it will not be possible to implement Mi in terms of Mi-1. Verifying the existence of such correspondence is part of the proof-of-correctness aspect of the HDM. Since the state of an abstract machine is represented by the values of its state variables, the mapping function, mfi, consists of a set of relations expressing each state variable of Mi in terms of the state variables of Mi-1.

As noted earlier, a finite-state machine is characterized by a set of operations and the changes they produce on the state

variables of the FSM. In trying to relate two FSM's to one another, we must relate their state variables and their operations. Therefore, we must next relate the operations of Mi to those of Mi-1. Since Mi is to be implemented in terms of Mi-1, the desired relations are a set of <u>programs</u>, pi-1, which implement each operation of Mi in terms of the operations of Mi-1.

The relationships between elements of the abstract machines in the two-level hierarchy are, therefore, as shown below.



Two independent <u>specifications</u> define the operations and state variables of Mi and Mi-1. If Mi is in state s1i and operation oi is performed, Mi will then be in state s2i. If Mi is to be implemented in terms of Mi-1, then the <u>mapping function</u>, mfi, relates each state, s1i and s2i, of Mi, to one or more states, s1i-1 and s2i-1, of Mi-1. Each operation, oi, of Mi is related to the operation of Mi-1 by a <u>program</u>, pi-1. More particularly, an operation, oi, of Mi is implemented by a <u>program</u>, pi-1, which is executed by Mi-1. The <u>mapping function</u> also indicates the manner in which the state of Mi is derivable from the state of Mi-1.

The <u>specification</u> for a finite-state machine (abstract machine) relates the state (values of state variables) of the machine prior to execution of an operation to its state after execution, for each operation. To prove the consistency of the <u>mapping function</u>, mfi, with the <u>specifications</u> for Mi and Mi-1, <u>mapped specifications</u> are produced for Mi. Since the mapping function, mfi, expresses each state variable of Mi in terms of the state variables of Mi-1, these expressions can be substituted in the specification for the operations of Mi. The result is a mapped specification of the operations of Mi in terms of the state variables of Mi-1 which should be proved consistent with the operations of Mi-1 as defined in the specification for Mi-1 (the operations for Mi-1 are already defined in terms of the state variables of Mi-1 in the specification).

The program, pi-1, which implements each operation of Mi on Mi-1 can be proved consistent with the specification of Mi and Mi-1 and with the mapping function mfi. Since the specification of an operation of Mi relates the values of the state variables of Mi before execution of the operation to those after execution, the mapped specification of an operation of Mi expresses the same relation but in terms of the state variables of Mi-1. Therefore, such a mapped specification represents an assertion as to the relation between the state variables of Mi-1 prior to execution of an operation of Mi with such variables after execution. The program, pi-1, for a given operation, is to be constructed so that this assertion is true. Such a program is then analyzed as follows. The mapped specification for Mi contains or implies any necessary assumptions as to the values of the state variables, Vi-1, prior to execution of the related pi-1. The program is examined to determine the effect of each statement in it on the state variables. If alternate paths exist in the program, the effect of each is evaluated. Presumably any alternate paths in the program will correspond to alternatives in the mapped specifications. At the exit from the program, the values of the state variables should bear the relation to their initial values described by the mapped specification.

Hierarchy of i+1 Abstract Machines

Finally, an entire system is defined as a sequence of i+1 abstract machines. The specification for the topmost machine defines the system for the user (a person or a program). He should be able to use the system with no knowledge of the activities of the lower i machines. The lowest-level machine is the hardware or the view of the hardware seen in terms of a specific programming language. The total system is represented by i+1 specifications, i mapping functions to relate the state variables of each successive pair of abstract machines, and i sets of programs to implement the operations of each of the topmost i machines on the machine below it. Proof-of-correctness should be able to be carried out for each specification and for the mapping functions and programs relating successive machines. SRI feels that even in the absence of the proofs, the structured design and implementation should produce a better and more reliable product than traditional methods. This view of a system as a sequence of i+1 abstract machines might be called View A and is shown in Figure 1.

PRACTICAL CONSIDERATIONS

Theoretical considerations as discussed above yield a sequence of abstract machines, each of which implements that above it.

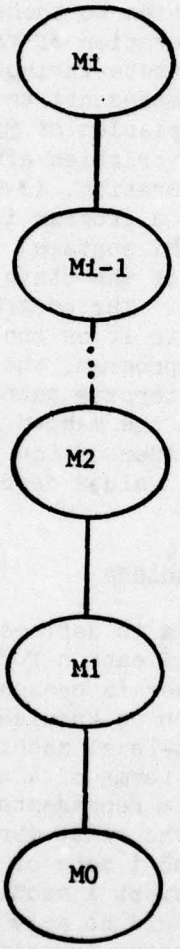


Figure 1. A Sequence of Abstract Machines (View A)

However, each such machine may be rather complex and duplicate some of the features of machines below it. Therefore, in practice, an abstract machine is modularized. The features of machine i which are different from those of all the machines below it are contained in one or more modules. In addition, machine i may contain a module corresponding to each module or portion of a module in machine i-1. Thus, each abstract machine could consist of more modules than the one on which it is implemented. However, it is also expected that the further apart two machines are in a hierarchy, the less likely it is that the features of the lower-level machine will be necessary or of interest to the higher-level machine. Otherwise, the features of the bottom-level programming language or hardware would still be part of the specification of the topmost machine as would the features of all the intermediate machines. This view of a system as a hierarchical set of (possibly) duplicated modules might be called View B and is shown in Figure 2. (In the figure it is assumed that only one new module is added at each level and that each module is visible throughout the hierarchy, that is, from its point of definition to the top.) The union of all the modules (mij, $0 \leq j \leq i$) present at a given level in the hierarchy in View B corresponds to an abstract machine (Mi) in View A.

A third view (View C) of a system is adopted which eliminates the duplication of modules by naming or pointing to the lower-level modules visible at a given higher level. This view is shown in Figure 3, which is similar to Figure 2 except that the only module shown at each level is that which contains features different from those of all modules at lower levels. Lower-level modules visible at higher levels are reflected as pointers. Since an abstract machine is treated as a collection of modules in Views B and C and since abstract machine Mi will be implemented on Mi-1, then Mi-1 must contain all the modules needed to implement Mi. Even though machine Mi-1 may not utilize a given module, mpp, for <u>its</u> implementation, mpp must be part of Mi-2 as well as Mi-1 if it is needed to implement Mi.

The principal basis suggested by the HDM for modularizing a software system is a practical one, from the design viewpoint, at least, whose various aspects have been given a variety of names. SRI talks of a module's being a "manager" for a particular type of object or system resource. It can be viewed as a collection of objects and the operations available on them. Parnas talks of "information hiding", whereby the design details of a particular portion of the system are contained in one module and hidden from other modules so that a design change is apt to affect a single module rather than ripple through a number of modules. Another pertinent term is "levels of abstraction" whereby a given module

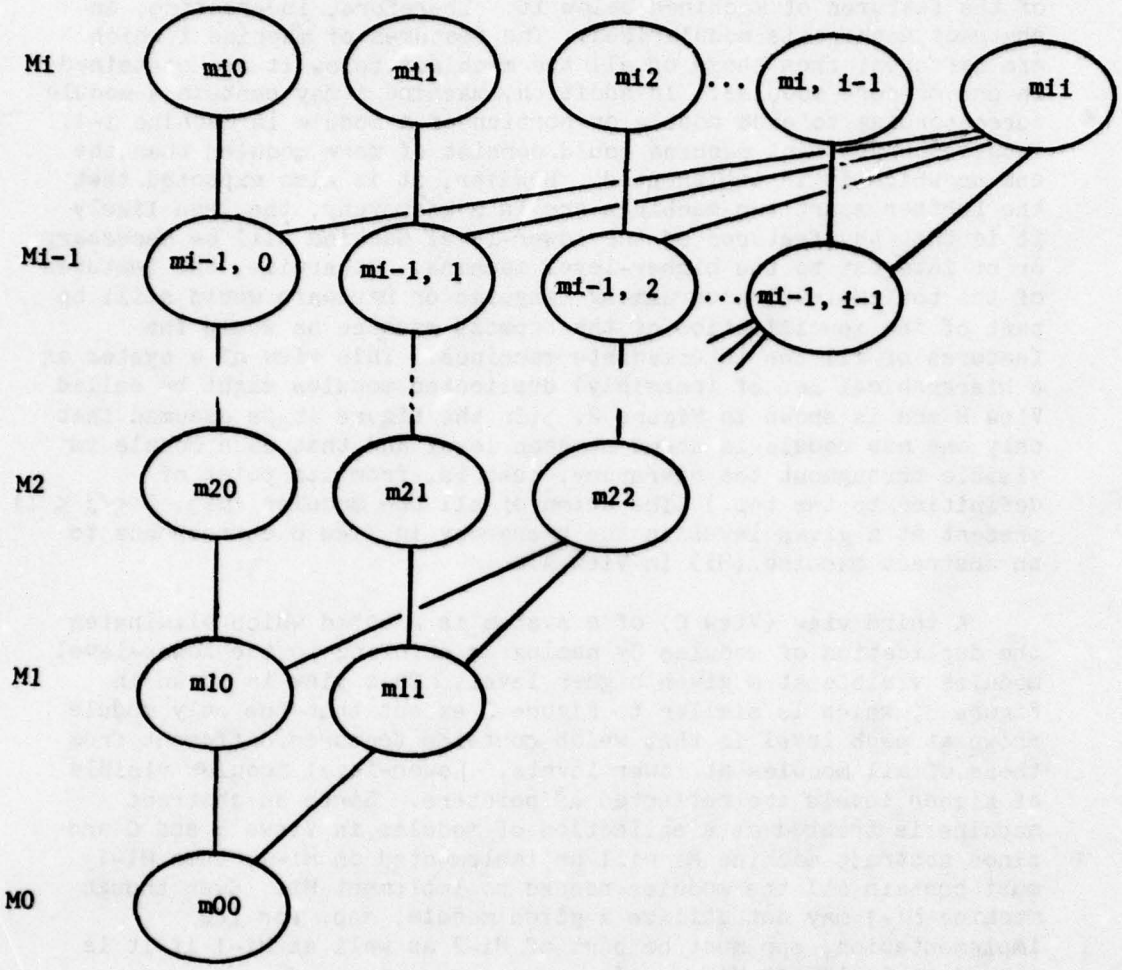
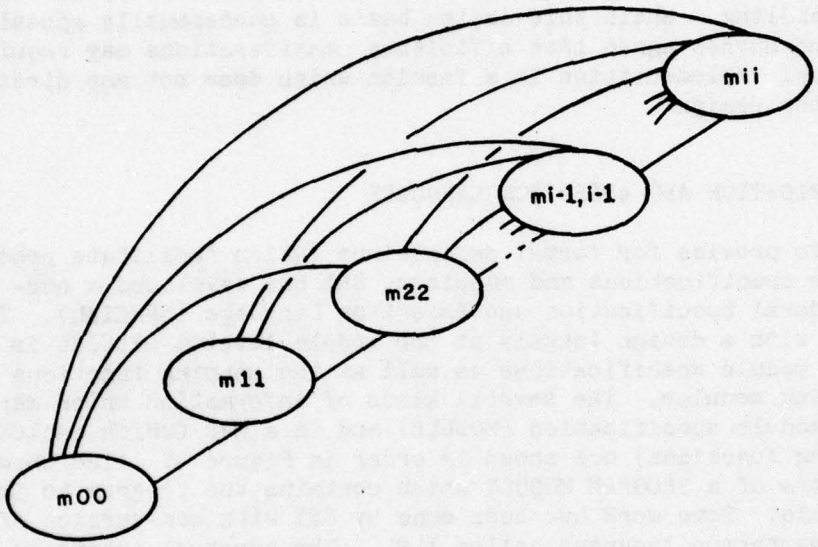


Figure 2. Hierarchical Set of Duplicated Modules (View B)



IA - 5,1524

Figure 3. HIERARCHICAL SET OF NON-DUPLICATED MODULES (VIEW C)

provides a particular level of data abstraction and operational abstraction. A lower-level module is less abstract, more concrete and specific in its view of data and operations. The general intent is to decompose a system into manageable modules, each of which is relatively consistent as to the level of its knowledge and actions, has sole responsibilities in a certain area, and supports higher-level modules which have more abstract duties and knowledge. In a message-handling system, one module might be responsible for byte handling and be dependent on a lower-level module responsible for bit handling. While this design basis is conceptually appealing, SRI and Parnas agree that efficiency considerations may require eventual implementation in a fashion which does not map directly onto the design.

SPECIFICATION AND ASSERTION LANGUAGE

To provide for formal definitions (which facilitate proofs) of module specifications and mappings, SRI has developed a non-procedural Specification and Assertion Language (SPECIAL). The HDM deals with a design largely at the module level. SPECIAL is used to write module specifications as well as the mapping functions relating modules. The several kinds of information which may appear in a module specification (MODULE) and in a MAP (which includes the mapping functions) are shown in order in Figure 4. Also shown is the form of a PROGRAM MODULE which contains the program to implement a module. Some work has been done by SRI with one version of an implementation language called ILPL. The eventual intent is to be able to compile machine code from programs written in an implementation language such as ILPL. At the present, SPECIAL is much further developed, with tools to support it (see below), than ILPL.

As Figure 4 illustrates, a MODULE and a MAP (as well as a PROGRAM MODULE) have various similarities. SPECIAL emphasizes its ability to deal with a considerable variety of (data) types. The TYPES paragraph (of a MODULE) contains definitions of the types used to describe the objects managed by the module, for instance. Variables of the module are declared and associated with a type under DECLARATIONS. PARAMETERS are symbolic constants used to specify capacities, for instance, and which are treated generally like variables in the design stages but which will be associated with specific values for a specific implementation. DEFINITIONS contain string-substitution macros used elsewhere in a specification to reduce the writing effort. EXTERNALREFS lists such things as data types and operations which are defined in other modules but referenced in this module. The ASSERTIONS paragraph seems to be

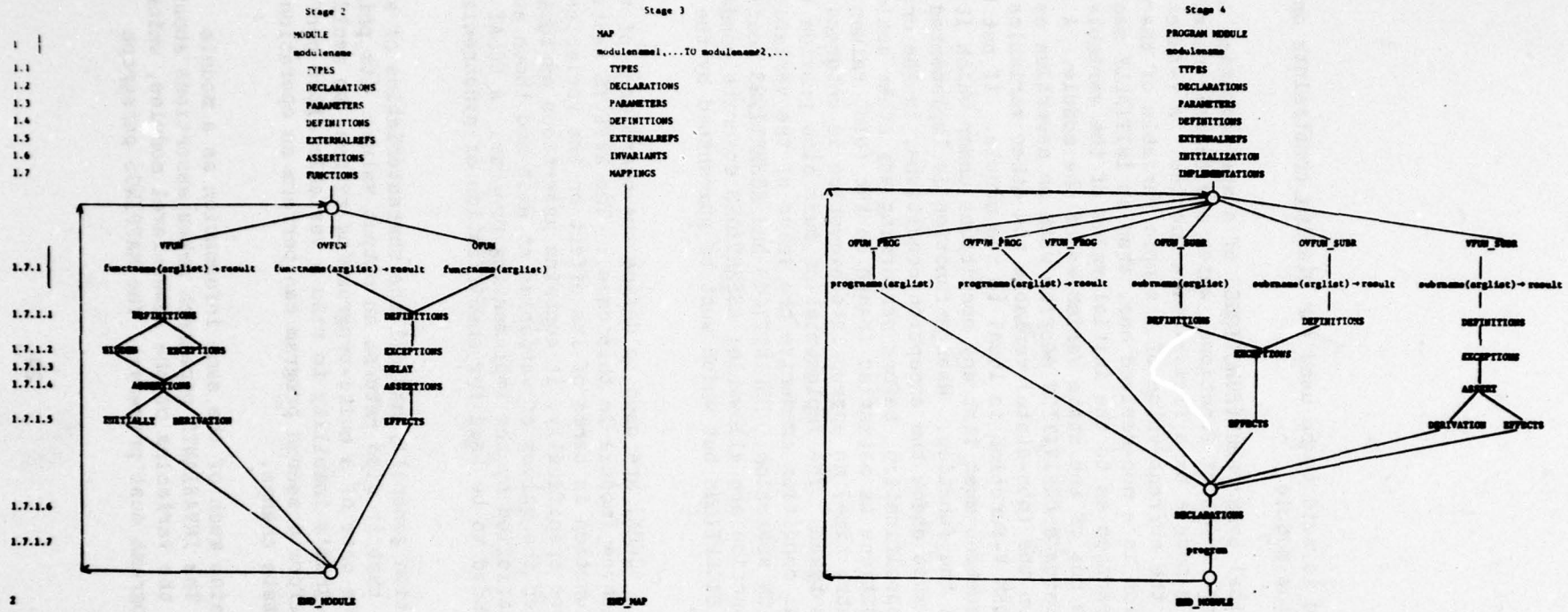


Figure 4. SRI "Specification" Contents

used rarely and is said to be used for placing constraints on the parameters of the module.

The principal paragraph (FUNCTIONS) of a specification is used to define several types of functions, which are closely related to the concept of a module as a finite-state machine. A V-function (VFUN) returns the current value of a single variable of the module. If the V-function is a non-derived one, then an INITIALLY section contains an assertion as to the initial value of the variable, and the variable is one of the state variables of the module. A derived V-function contains a DERIVATION section with an assertion as to the relation between the (non-state) variable and other variables of the module. A HIDDEN V-function is local to the module. If not hidden, an EXCEPTIONS section must list any conditions under which it is illegal to call the function. When a function is implemented, its implementation must check the exception conditions, in the order listed in the specification, before performing any other action. If one of the conditions is satisfied (based on the (old) values of the variables at entry time) an appropriate exception is returned and no other action is taken. The implementation must also provide the asserted initial condition or derive the value of the variable based on the DERIVATION assertion. DEFINITIONS and ASSERTIONS sections local to the function are allowable; ASSERTIONS describe conditions similar to the EXCEPTIONS but which must be guaranteed by the caller.

O-functions (OFUN) are used to define the operations of the finite-state machine (module in this case). The EFFECTS section defines the O-function in terms of its effect on the variables of the module. More specifically, it contains assertions as to the relations between the values of variables at exit and those at entry which must be satisfied by the implementing program. A DELAY section is intended to be used for specification of synchronization requirements.

An OV-function generally has all the characteristics of an O-function except that it also returns an output value. Its principal purpose is in the case of a multi-programming system, to avoid the problem of a program's inability to read a variable after performing an operation, before a second program can perform an operation and cause further state change.

A MAP contains much of the same information as a module specification. The INVARIANTS paragraph makes assertions about relations among the variables of the lower-level modules, which the implementing programs must preserve. The MAPPINGS paragraph

expresses the V-function mappings between a module and one or more lower-level modules that implement it.

TOOLS

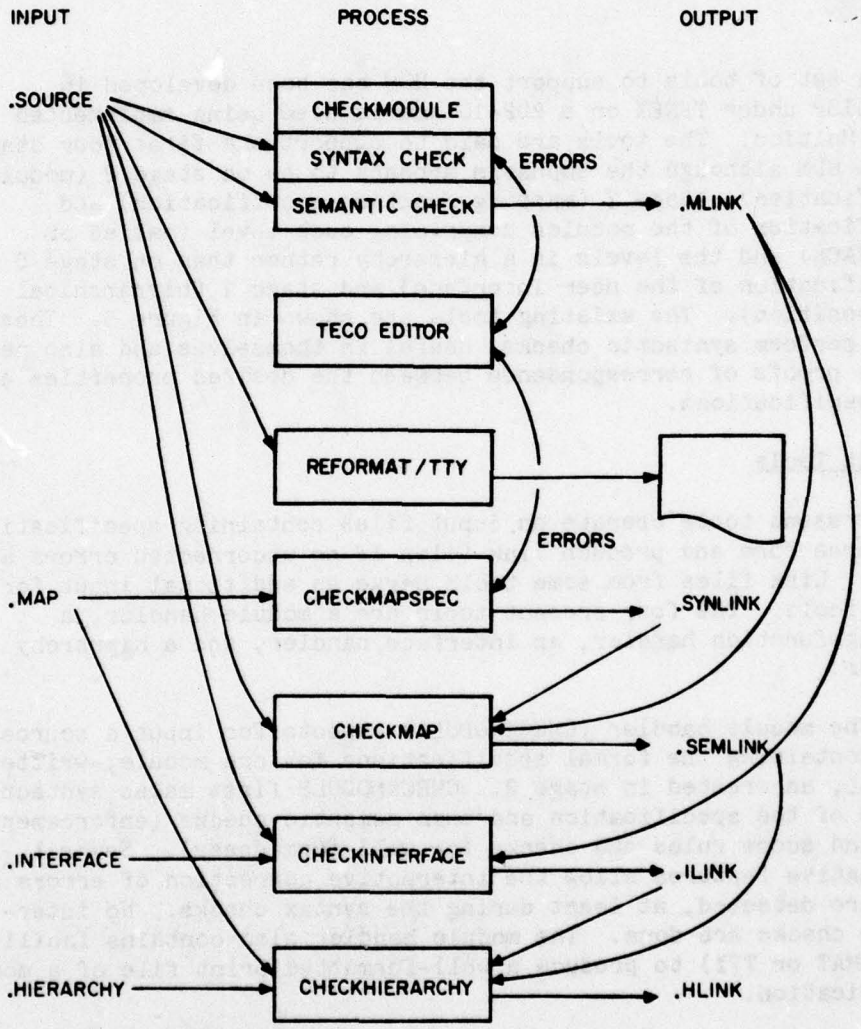
A set of tools to support the HDM has been developed in INTERLISP under TENEX on a PDP-10 and is also being implemented under Multics. The tools are said to support the first four stages of the HDM although the emphasis appears to be on stage 2 (module specification), stage 3 (mapping-function specification) and specification of the modules comprising each level (called an INTERFACE) and the levels in a hierarchy rather than on stage 0 (specification of the user interface) and stage 1 (hierarchical decomposition). The existing tools are shown in Figure 5. These tools perform syntactic checks, useful in themselves and also needed in the proofs of correspondence between the desired properties and the specifications.

Present Tools

Present tools operate on input files containing specifications in source form and produce link files if no uncorrected errors are found. Link files from some tools serve as additional input for other tools. The four present tools are a module handler, a mapping-function handler, an interface handler, and a hierarchy handler.

The module handler (CHECKMODULE) expects for input a source file containing the formal specifications for one module, written in SPECIAL, as created in stage 2. CHECKMODULE first makes syntactic checks of the specification and then semantic checks (enforcement of type and scope rules and checks for well-formedness). Several alternative features allow the interactive correction of errors as they are detected, at least during the syntax checks. No inter-module checks are done. The module handler also contains facilities (REFORMAT or TTY) to produce a well-formatted print file of a module specification.

The mapping-function handler is split into two pieces. CHECKMAPSPEC expects for input a source file containing one map between modules of two adjacent levels, written in SPECIAL, as created in stage 3. Internal consistency checks are made as in the case of the module handler. External consistency checks are made by CHECKMAP which uses the link file from CHECKMAPSPEC for its primary input. The checks are directed principally toward determining that a map contains one and only one (proper) mapping-function expression



IA-5,1525

Figure 5. SRI Tools

for each primitive object (parameter, designator, or non-derived V-function) of an upper-level module.

The interface handler (CHECKINTERFACE) expects for primary input a source file containing one interface (level) specification which names the interface and lists the modules comprising the interface. The specification is written in an extension to SPECIAL defined in BOYE76. Checks are made of the well-formedness of the specification and that module specifications for the listed modules exist and have been checked and that EXTERNALREFS for each module reference only other modules in the list.

The hierarchy handler (CHECKHIERARCHY) expects for primary input a source file containing one hierarchy specification. A hierarchy, in conjunction with the previous files discussed, is intended to describe a complete software system design or at least n successive levels of such a design. A hierarchy relates successive pairs of levels (interfaces), from lowest to highest (the lower level implements the higher one of a pair), and names the maps relating the modules of each upper level with those of the (next) lower level. The hierarchy specification is written in another extension to SPECIAL defined in BOYE76. Checks are made of the well-formedness of the specification, that the interfaces and maps listed exist and have been checked, and that the mapping-function specifications are consistent with the interface specifications.

Future Tools

In addition to the current tools, a number of other tools are planned, contemplated, or under development. These additional tools, which support implementation and proofs, are:

1. A model consistency checker for designs involving semantic dependencies in the proofs of correspondence between the desired properties and the specifications. It performs extrinsic syntactic checks and "generates logical formulas whose validity is equivalent to the satisfaction of the semantic conditions for consistency with the model". The generation of the logical formulas is said to be straightforward. The formulas would be proved by hand or with machine assistance to reduce human error.

2. A program handler to check program syntax and syntactic consistency of a program with the specifications and mapping functions.

3. A development data-base manager to maintain a data base of the specifications, programs, and proofs and to record which modules have been specified, mapped, implemented, and verified.

4. Other tools to support semi-automatic program verification for use as appropriate:

   a. Verification condition generator for implementation programs.

   b. Logical simplifiers.

   c. Various program transformers (e.g., optimizers that preserve program equivalence).

   d. An implementation proof checker.

DESIGN AND IMPLEMENTATION METHODOLOGY

The HDM suggests that system development be done in five approximately sequential stages. The proofs possible for most stages are not discussed in this section. The five development stages are as follows.

Stage 0 Design - Interface Definition

The top-level (user) interface is "defined", from the vantage point of View A (Figure 1). For this stage, "defined" is interpreted to mean that the state variables of the top-level abstract machine and the operations available upon it are determined, rather than formal definition. The interface is then decomposed into a set of modules in the fashion of View B. Although the interface is the top-level one and the modules will be top-level ones in View B, they will not necessarily remain top-level ones in View C after stage 1 is completed nor do they as yet comprise a complete system. At this stage the modules included are those implied by the user requirements. Each of these modules can be considered a facility which manages objects of a particular type and that is the basis for the decomposition of the interface. Available at the completion of stage 0 is an outline or sketch of (possibly only parts of) some of the modules of the system.

Stage 1 Design - Hierarchical Decomposition

Additional modules which are necessary to support those visible at the user interface but are hidden from the user interface are sketched. ("Sketched" is intended to mean that the V-, O-, and OV-functions of each module are listed in function form, such as create_object(t) -> c. As indicated in Figure 4, any function may have input arguments and V- and OV-functions have a result argument.) In addition to this sketching of the principal parts of each module, it is decided at which level (according to View C, Figure 3) each module should be defined and at how many higher levels each module or individual function of a module should be visible. Thus, the modules of each level are determined, which means that the modules at level i-1 upon which level i will be implemented must be determined for the entire hierarchy (for all values of i).

The modules sketched in stage 0 were based on the user interface. If some of these modules are defined at levels below (but available at) the user interface because they are part of the basis for implementing higher-level, visible or non-visible modules, they may need to be expanded to include further related functions which are not visible at the user interface. It seems likely that in most cases the steps of stage 1 will have to be performed iteratively. In evaluating whether each level has an adequate implementation base, modules may need to be moved up and down in the hierarchy. For such modules, it might be necessary to combine, split, or add new functions or delete ones no longer needed. Such changes might also include adding new modules or deleting old ones. If such changes are necessary in stage 1, then it may also be necessary to loop back to stage 1 from later stages since the later stages provide formal or informal proof of the adequacy and goodness of the design partitioning done in stage 1.

Available at the completion of stage 1 is at least an initial outline of most or all of the modules of the system in addition to at least an initial structuring of the system. The information is available to construct the INTERFACE and HIERARCHY files discussed under Tools although the use of the tools to check them is not possible at this time.

Stage 2 Design - Module Specification

In this stage a formal specification is written in SPECIAL for each module in the design. The format and content of a specification are shown in the MODULE column of Figure 4. The

module handler discussed under Tools can be used to check and to print the specifications.

A module is intended to have sharply defined responsibilities which do not intersect those of other modules. A specification is also intended to be independent of implementation decisions. Although a specification for a module is intended to be independent of those for other modules, SPECIAL provides the EXTERNALREFS paragraph in a specification to indicate dependencies that exist. The degree of dependence of a module on other modules is indicated by the number of other modules and their objects listed in the EXTERNALREFS paragraph.

Stage 3 Design - Mapping Functions

Although a MAP can contain a number of paragraphs similar to those of a module specification (see Figure 4), the most important one is MAPPINGS, which is used to relate the states of modules to those of lower-level modules. In a specification, the state variables correspond to the non-derived V-functions of the module, and the state of the module corresponds to an n-tuple containing the current value of each state variable. The mapping functions in a MAP relate each state variable of the upper-level module to an expression containing the state variables of lower-level modules. Thus, the design assumptions made as to the relationships between the state variables of modules at different levels are explicitly recorded.

Stage 4 - Implementation

The intent is that programs written in stage 4 be directly compiled into executable code. SRI speaks of a programming language which fulfills the functions of an abstract programming language as well as those of an implementation language.

The purpose of this stage is to implement the design specified in the preceding four stages. Programs must be constructed which implement the specifications from stage 2 and the mapping functions from stage 3. The exception conditions listed in the specification for each function need to be tested in the order listed. If an exception condition is true, then the program must return an exception to the higher-level program and return no value and perform no effects. If a function has n exception conditions, then its program will have n+1 exits.

If no exception conditions are true, the program for a V-function must produce an output which satisfies the appropriate

mapping function. For an O- or OV-function, the program must guarantee that at exit the state variables bear the relationship to their values at entry that is expressed in the EFFECTS section of the specification. The result returned by an OV-function seems to be specified also as part of the EFFECTS.

From the view of a sequence of finite-state machines, the above steps implement (the state variables and operations of) machine Mi in terms of the state variables and the operations of machine Mi-1. The "abstract programming language" is used to write these programs. The other step of implementation is to implement the primitive functions of the bottom-level module(s) as well as the primitive functions of the abstract programming language in the "implementation language". The form of communication between levels must also be determined, such as macro expansion or procedure calls. SRI admits that in some cases efficiency considerations may require that implementation not be based so closely on design as described above.

EXAMPLE

The stages of the SRI methodology, as described in Neum77, are carried out below for a simple example system. The principles of the design and proof methodology will be illustrated, even though all checking and proofs will be done manually. Acquaintance with the fundamentals of the specification language is assumed below.

SRI uses mixed conventions to refer to the result produced by execution of a V-function or OV-function. SRI defines a V-function in the form "VFUN seg(loc)" -> data" and then references the result as "data" or "seg(loc)", depending on context. In the following example, a more consistent notation replaces the three SRI references with "VFUN seg(loc)", "seg", or "seg(loc)", respectively.

Design and Implementation

The example system (see Figure 6) is a random access storage area called a **segment**, whose current size is limited by a fixed maximum. Stage 0 lists the functions in the user interface of this system. The V-function **seg**(loc) returns the item in location loc of the segment. Locations range from zero to some implementation-defined upper bound, but only locations less than the value of another V-function **size** are considered to be in the segment. There are two O-functions: **rewrite**(loc,data) to update a location in the segment with a given data item, and **append**(data), to add a new item at the end, increasing **size** by one.

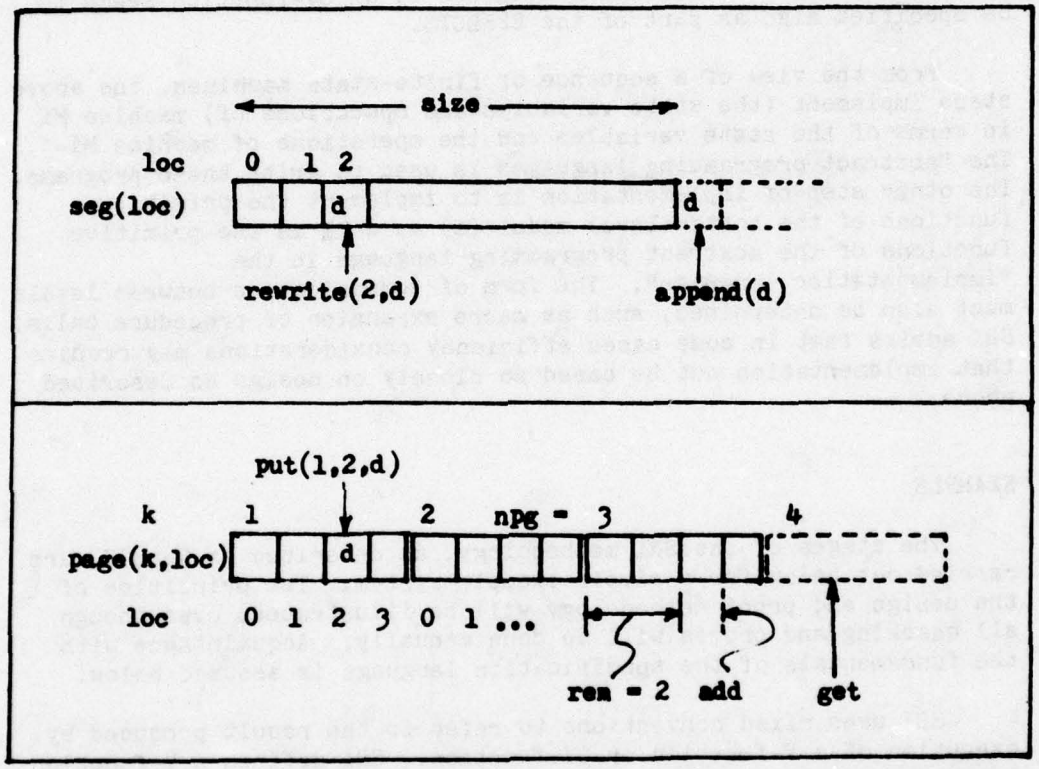


Figure 6. segment/page System

Level Number	Level/Module Name	V-function	O-function		
2	segment	seg(loc)	rewrite(loc,data)	stage 0	stage 1
		size	append(data)		
1	page	page(k,loc)	put(k,loc,data)		
		ngp	get		
		rem	add		

Note: In the example, stage 3 causes iteration of stages 1 and 2 to add two derived V-functions, pgn(loc) and dis(loc), to the page module.

Figure 7. Products of Stages 0 and 1

Figure 7 has been added to the Millen figures by the author of this report to emphasize the assumed form and contents of the products from stages 0 and 1. In a more complicated system, the user interface from stage 1 might be different from stage 0 - it might be split into several modules, some of which might be defined at lower levels for use by intermediate levels as well as the user interface.

In stage 1 a hierarchical design of the system is sketched. To keep the example small, we have just two levels: the user interface or segment level, level 2, and one lower level, called the page level, level 1. See Figure 6. The lower level system allocates storage in pages of fixed size. It has a page-read function page(k,loc) indexed by page number and displacement, an update function put (k,loc,data) for rewriting one location, and a function get to obtain a new page.

The segment envisioned in the user interface is implemented by several consecutively numbered pages, ending somewhere within the last page. A function npg gives the length of the segment in pages. A function rem is also needed to keep track of the size of the fractional part of the segment within the last page, and a function add is needed to update rem.

All the functions in the user interface level are put together into one module, and all the page level functions are assigned to a second module implementing the first. More than one module per level is not needed in this example since there is only one data structure at each level: the segment at level 2 and the page array at level 1.

For stage 2 we provide a formal specification in Figure 8 for the functions listed above. (The SRI convention is observed of denoting new values of V-functions by preceding the V-function name by a single quote symbol. Unquoted V-function names denote old values.) These specifications illustrate some, but not all, of the features of SPECIAL. To avoid cluttering the specifications, no type specifications are shown. All parameters, arguments, and values are assumed in this example to be the same type: non-negative integers with some implementation-defined upper bound.

One of the resource parameters of the system is the desired maximum segment size maxsize. The size bound is implemented in the page level with two bounds: maxnpg, the upper bound for the number of pages, and pgsize, the page size.

```
MODULE segment

     PARAMETERS

maxsize

     FUNCTIONS

VFUN seg(loc)

VFUN size

OFUN rewrite(loc,data)

     EXCEPTIONS
          loc > size -1

     EFFECT
          'seg(loc) = data

OFUN append(data)

     EXCEPTIONS
          size = maxsize

     EFFECT
          'seg(size) = data
          'size = size + 1
```

```
MODULE page

     PARAMETERS

maxnpg

pgsize

     FUNCTIONS

VFUN page(k,loc)

VFUN npg

VFUN rem

OFUN put(k,loc,data)

     EXCEPTIONS
          k > npg
          k = npg AND loc > rem -1

     EFFECTS
          'page(k,loc) = data

OFUN get

     EXCEPTIONS
          npg = maxnpg

     EFFECTS
          'npg = npg + 1
          'rem = 1

OFUN add

     EXCEPTIONS
          rem = pgsize

     EFFECT
          'rem = rem + 1
```

Figure 8. Specifications for segment Module and page Module

Stage 3 provides the mappings between consecutive levels. Each V-function in the segment level is expressed as a function of page level V-functions in Figure 9. In order to express the seg mapping, it was convenient to introduce two derived V-functions to be added to the page level: pgn, which gives the page number for a segment location, and dis, giving its displacement within the page. The arithmetic expressions given are written, not in pure mathematical notation, but in SPECIAL, whose expressions are upwardly compatible with ILPL. Thus, the division (/) of two integers has an integer result, the integer part of the real quotient.

The implementation of each segment level O-function as an "abstract" ILPL program calling page level O-functions, as shown in Figure 10, is the product of stage 4. All V-functions other than bottom level primitive V-functions must also be implemented with programs. The programs implement derivations, in the case of derived V-functions, or mappings, in the case of higher level primitive V-functions. Bottom level primitive V-functions end up as program variables in the implementation.

Exception conditions arising from function calls are reported, according to the semantics of SPECIAL, by setting a "meta" function ERRORCODE( ) to the character string identifying the exception in the specification. The ILPL user can identify which exception has occurred, if any, as a result of the function call, by using the DO...WITH construct. In this example there is no need to distinguish between exceptions, so the DO...WITH, and also the ILPL RETURN statement, which sets the higher level ERRORCODE, are not used.

Note that the implementation for rewrite does not do any exception testing itself, but lets the call on put do the testing. If an exception occurs, put has no effect, and hence neither will rewrite. The program for append, on the other hand, does enough testing to ensure that the called functions will not encounter any error conditions.

Since the page level is the bottom level in our design, each of its O-functions must either be a built-in function of ILPL, or have an ILPL subroutine implementing it. Trivial subroutines for put, get, and add are shown in Figure 11. Now all stages of the design and implementation are complete.

Verification

Stage 0' of the verification consists of stating the required properties of the user interface. For this example we might want to

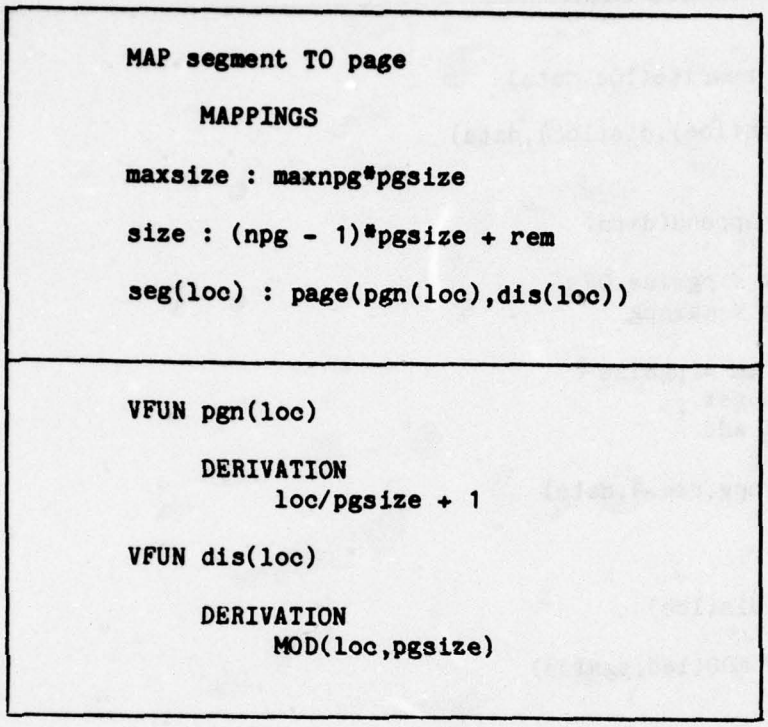


Figure 9. segment/page Mappings and Supplement to Specifications

```
OFUN_PROG rewrite(loc,data)
  BEGIN
    put(pgn(loc),dis(loc),data)
  END

OFUN_PROG append(data)
  BEGIN
    IF rem < pgsize OR
       npg < maxnpg
    THEN
      IF rem = pgsize
      THEN get
      ELSE add
      FI;
      put(npg,rem-1,data)
    FI
  END

VFUN_PROG dis(loc)
  BEGIN
    dis <- MOD(loc,pgsize)
  END

VFUN_PROG pgn(loc)
  BEGIN
    pgn <- loc/pgsize + 1
  END

VFUN_PROG seg(loc)
  BEGIN
    seg <- page(pgn(loc),dis(loc))
  END

VFUN_PROG size
  BEGIN
    size <- (npg - 1)*pgsize + rem
  END
```

Figure 10. Implementations

```
OFUN_PROG put(k,loc,data)
  BEGIN
    IF k < npg OR (k = npg AND loc ≤ rem -1)
    THEN page(k,loc) <- data
    FI
  END

OFUN_PROG get
  BEGIN
    IF npg < maxnpg
    THEN
      npg <- npg + 1;
      rem <- 1
    FI
  END

OFUN_PROG add
  BEGIN
    IF rem < pgsize
    THEN rem <- rem + 1
    FI
  END
```

Figure 11. ILPL Implementations of Bottom-Level O-Functions

check a condition such as "a seg(loc) reference to a location modified in the past by rewrite(loc,data) or append(data) yields the same data as long as any intervening rewrite calls refer to different locations." A simpler condition, more suitable for a short example, would be:

$$size \leq maxsize. \qquad (1)$$

Stage 1' of the verification just checks that function names are not duplicated in different modules and that the hierarchical structure contains no loops - for example, the page module should not be implemented by the segment module.

At stage 2' the specifications of the functions in each module are available, permitting internal checks within each module. Having the user interface specification in particular, we can check (1). This condition is a global assertion; it is proved inductively by showing that the initial state satisfies it, and every O-function preserves it.

In our example we have not yet specified an initial state; that would be done when the system is finally put into operation. The initial state will be required to satisfy all desired global assertions, as well as any INITIAL conditions specified in V-functions. To increase our confidence that the system will be of some use, we can exhibit a possible initial state now; this will establish the logical consistency of the global assertion.

The state

$$maxsize = n$$
$$size = 0$$
$$seg(loc) = 0 \text{ for all } loc$$

satisfies (1), if n is an arbitrary non-negative integer in the implementation range.

Every O-function preserves (1), since rewrite does not modify size, and append has no effect once size reaches maxsize.

Other global assertions can be stated and proved for the page module. One cannot tell what global assertions are needed at lower levels until the stage 4' verification of the implementation is in progress. Looking ahead, we find that two global assertions will be used:

$$npg \leq maxnpg \quad (2)$$

$$rem \leq pgsize \quad (3)$$

They are obviously consistent and preserved by each of put, get, and add.

At stage 3' we have the mappings and can check that they are consistent. This is done by using them to map each upper level O-function effect down to its lower level equivalent and showing that the lower level effect is consistent. This check is essentially to prevent independent state variables from being mapped down into the same lower level state variable. For example, the mapping:

a : b

c : b

would be inconsistent if there were an upper level function that changed a differently from c, say with the effect:

'a = 1

'c = 0.

Then the mapped effect would be:

'b = 1

'b = 0

which is inconsistent.

The expressions tested for consistency in this step appear as output assertions in the implementation programs. Hence, testing for inconsistency this way is redundant, as SRI points out, since the implementation proofs in stage 4' would fail in the presence of any inconsistencies. The purpose of stage 3' is to save us the trouble of attempting stage 4' when mapping mistakes doom it to failure. Since it is tedious to do consistency checking manually, let us, in this example, go on to stage 4'.

In stage 4' we verify the correctness of the programs. The V-function programs are trivial, since they implement derivations or mappings that have already been expressed in ILPL-compatible notation.

There are two 0-function programs, for rewrite and append. Figure 12 shows the analysis of rewrite in the form of a flow chart with assertions at points or "nodes" between the statement boxes. This is not the way SRI does it; their approach is suited to semiautomatic verification with their software tools. The proof illustrations below employ a manual technique that is based, however, on the Floyd method that also forms the basis for SRI's system.

Our job is to show that if the exception condition holds, the program has no effects; and if the exception condition does not hold, the specified effects occur. Two paths are indicated through the flow chart, corresponding to whether or not the exception condition of the rewrite specification is true. Assertions in parentheses are relations among upper level V-functions.

The first assertion, in parentheses, on node 0 of path 1, is the upper level statement that the exception holds; it is an input assertion. The same assertion then appears in its mapped down form. The next two assertions on the same node are logical consequences of the first. Their purpose is to establish that one of the exception conditions holds in the following call of put. In general, the starred assertions in the flow chart are those that are not just the result of a substitution, and hence require proof. Only two such proofs will be illustrated, one for each function,* because they are easy in this example. They do, however, represent details that have to be right, so it is worthwhile to have done the checking.

When put is called in path 1, it will have no effect, because one of its exception conditions holds. This fact is written as an output assertion on node 1 of path 1.

On node 0 of path 2, the consequences of the input assertion imply that neither exception condition of put holds, and we may then take the effect specified for put as an assertion for node 1, just after the call. Checking the mapping for seg, we find that this assertion is exactly the effect of rewrite, as desired.

The analysis of append is shown in Figure 14. The three paths are distinguished by the tests in the program, but the final assertion on path 1 shows that this path corresponds to the exception condition of the append specification. In the other two paths it is shown before each call that no exception conditions hold for the lower level functions called, and hence their effects are

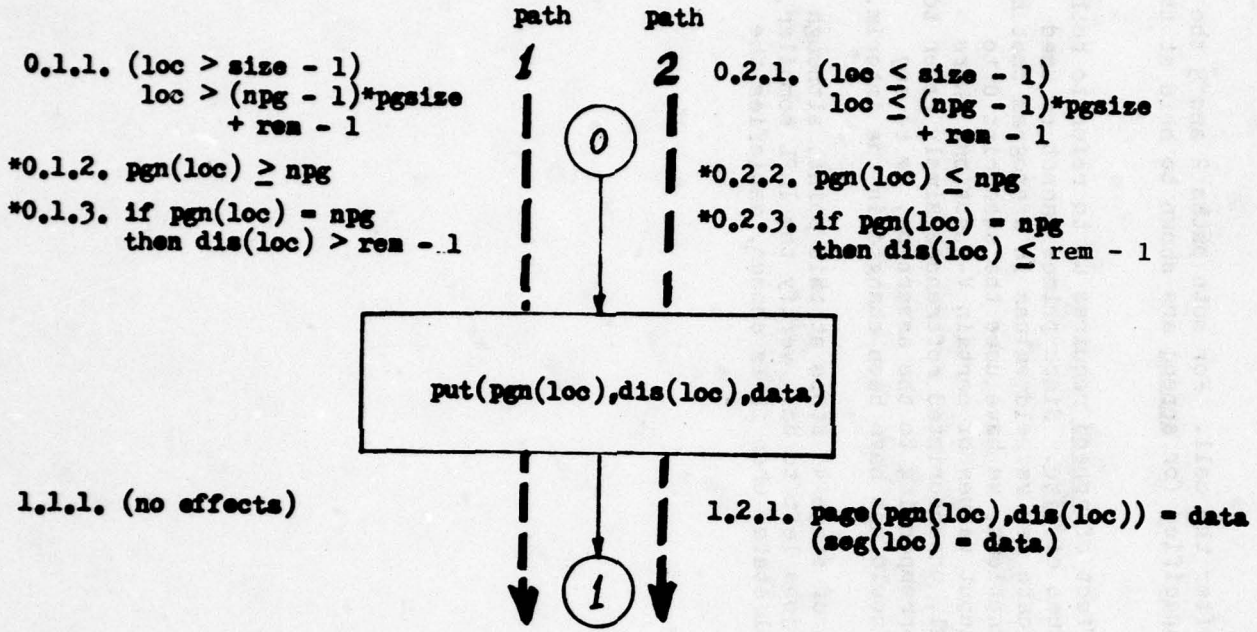
---

* In Figures 13 and 15.

taken as assertions after the call. For both paths 2 and 3 the upper level effects specified for append are shown to hold at the exit node, 6.

Note that the effect of append requires us to refer to both initial and final values of size. Since primes cannot be used unambiguously to indicate new vs. old values in a program that may call two or more O-functions, we have used the subscript 0 to indicate initial or input values of certain V-functions where necessary. In general, unsubscripted references simply refer to the values at the node corresponding to the assertion, or to any previous values that could not have been changed in the interim.

The verification of stage 4' stops at this point, although there are really two jobs left to do: verify the ILPL compiler, and check that the initial state, when it is chosen, satisfies the desired conditions.



Key to numbering: node.path.assertion

Figure 12. Analysis of rewrite Program

LEMMA 0.1.2 (rewrite)

PROOF

pgn(loc) = loc/pgsize + 1 (derivation)

$\geq$ npg - 1 + (rem-1)/pgsize + 1 (0.1.1)

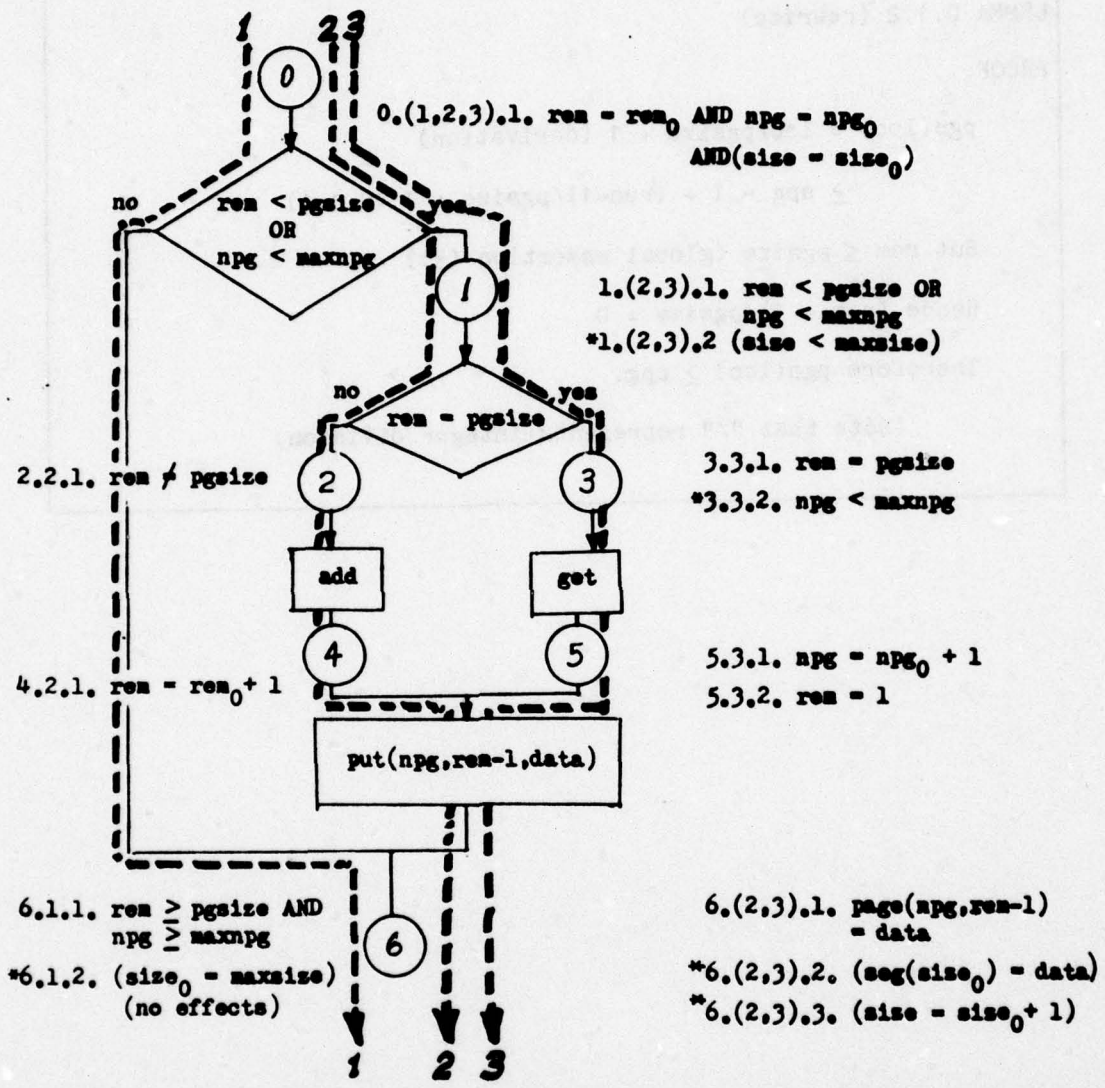
But rem $\leq$ pgsize (global assertion (3))

Hence (rem - 1)/pgsize = 0

Therefore pgn(loc) $\geq$ npg.

(Note that "/" represents integer division)

Figure 13. Sample Proof for rewrite Program Analysis



Key to numbering: node.path.assertion

Figure 14. Analysis of append Program

```
LEMMA 6.3.2 (append)

PROOF

     pgn(size ) = size /pgsize + 1   (derivation)
             0         0

                = npg   - 1 + rem /pgsize + 1   (mapping)
                     0           0

                = npg   + 1   (3.3.1)
                     0

     dis(size ) = MOD(size ,pgsize)   (derivation)
             0              0

                = MOD((npg   - 1)*pgsize + rem ,pgsize)   (mapping)
                           0                      0

                = MOD(rem ,pgsize)
                         0

                = 0   (3.3.1)

     seg(size ) = page(pgn(size ,dis(size ))   (mapping)
             0                 0          0

                = page(npg +1,0)   (by the above)
                           0
                = page(npg,rem-1)   (5.3.1 and 5.3.2)

                = data   (6.3.1)
```

Figure 15. Sample Proof for append Program Analysis

SECTION IV

HONEYWELL METHODOLOGY

Several related efforts within Honeywell, Inc., have contributed to the methodology discussed here. The initial efforts were by a group of people including R. C. McGee and A. Pizzarello at Honeywell Information Systems in Phoenix. A methodology was evolved which has been referred to as a set of Rational Programming Techniques. The methodology is based on work by E. W. Dijkstra and C. A. R. Hoare (Dijk75, Dijk76, Hoar69, Hoar72b). As the methodology was evolved, it was applied to development of a Virtual Machine Monitor (VMM), which allows several operating systems to be run concurrently on the same computer, as well as to other smaller projects.

More recently the methodology from Phoenix has been referred to as the WELLMADE methodology. It has been studied by D. L. Boyd and others at Honeywell, Inc., in Bloomington, Minn. Boyd and G. J. Gustafson have discussed their view of WELLMADE in Boyd76b.

At the present time, Boyd and Stanley Vestal are developing a Rational Design Methodology (RDM) under Honeywell contract to RADC (RADC contract number F30602-77-C-0043). The RDM generally consists of a superset of the features of WELLMADE.

The Honeywell methodology discussed here is intended to be the RDM; however, documentation of the RDM is rather minimal as yet. Documentation of WELLMADE is somewhat more extensive, although the only publicly available documentation found on any of the work mentioned above is Boyd76. Consequently, this discussion is heavily based on informal documentation of the Phoenix, WELLMADE, and RDM efforts and on personal communications with Boyd and Vestal.

GENERAL CONCEPTS AND DESIGN DOCUMENTATION

The basic unit for software design in the RDM is the module. As in the case of SRI, the module is viewed as an abstract machine or finite-state machine which is characterized in terms of its data structures (state variables) and programs (operations upon them). Lower-level modules form a base for higher-level ones. In addition to the modular hierarchy, each program within a module may be decomposed in terms of internal program blocks which can also be

decomposed in terms of internal program blocks, etc., to the extent desired.

The structure for documentation (including a language for formal specification) of a design is specified for the RDM in BNF (Backus-Naur form). The overall documentation structure is shown in Figure 16. The line numbers in the column at the left and the block numbers are not part of the BNF specification but have been added to simplify this discussion. For further simplication in Figure 16 and similar ones which follow, the leading portion of the block numbers (1.2.2.2.) has been dropped for most blocks -- particularly those within the two outlined areas. In most (but not all) cases, blocks which are nodes but not leaves contain no information but simply provide linkage. The intent is that the entire design documentation (all of the contents of Figure 16) be stored in a computer as the design is evolved.

Figure 16 represents the design for a complete software system. Block 1.1 is an index of the modules of the system. The remainder of the figure, starting with block 1.2, is repeated to the extent necessary for each module in the system. Blocks of information are delineated by the use of a number of reserved words (entered in the computer in an underlined form and generally shown in that form in this report), and other similar words are used within certain blocks. The BNF not only specifies the design documentation structure but also the syntax of a program design language (PDL).

Several of the blocks in Figure 16 are to contain unformatted material denoted as "text". Such blocks in line 4 are used for English-language descriptions of certain aspects of each module. Based on WELLMADE documentation, these blocks have the following uses. The "functional-description" block describes the function or purpose of the module. The "usage-information" block tells a user how to use the module and at least those for higher-level modules are used to provide the basis for user's manuals. The "acceptance-criteria" block describes resource utilization objectives such as time (performance) or space. The "design-overview" block describes how the module fulfills its function, the way it operates. The purpose of the "notes" block on line 4 (and on lower lines) is apparently to record design alternatives and the reasons for selecting a particular one as well as limitations of the design which may need to be taken into consideration at implementation time.

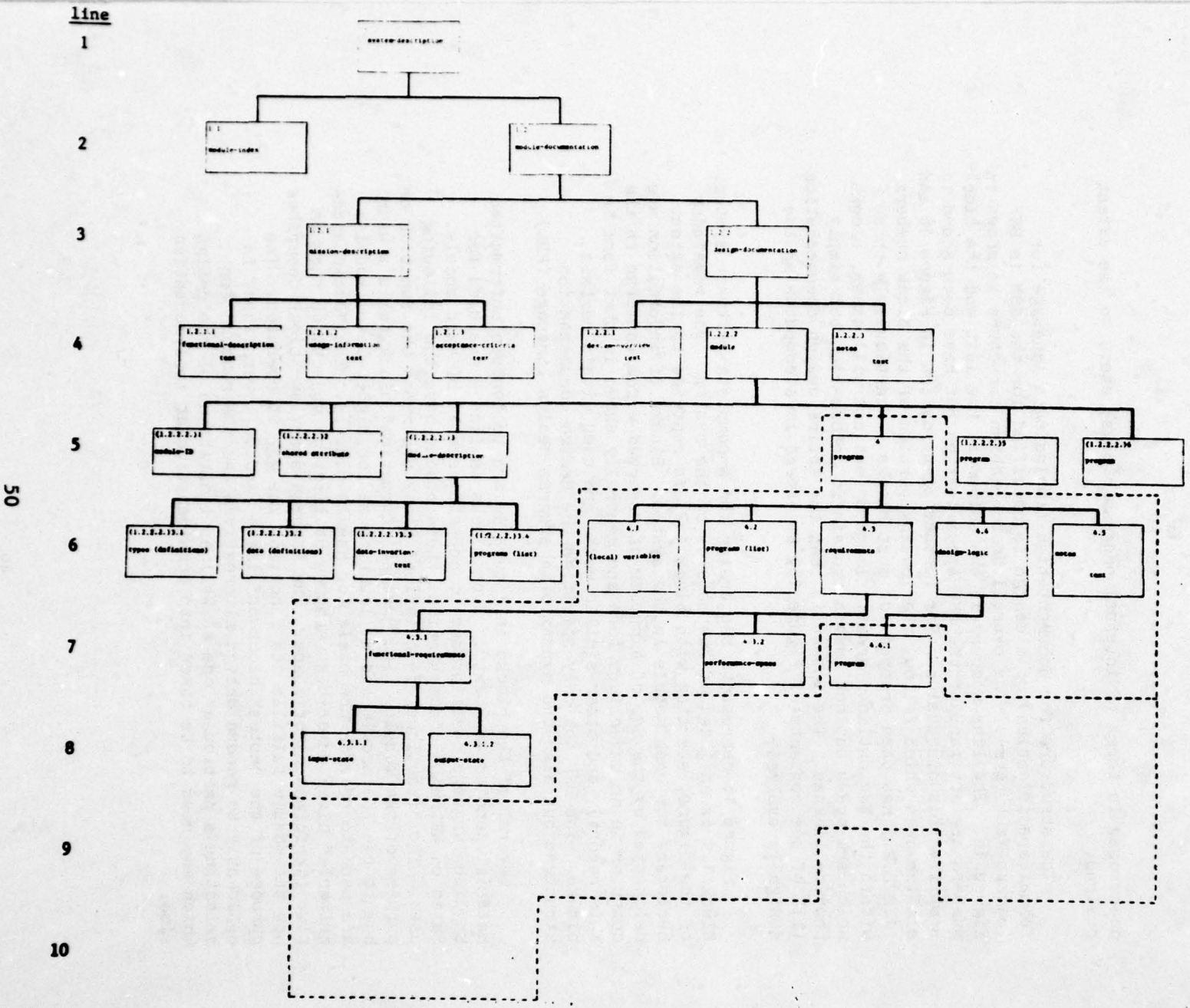


Figure 16. Design Documentation Structure

SYSTEM STRUCTURE AND DESIGN PHASES

Two design phases - general and detailed - are grossly associated with different parts of the structure. A general design phase involves specification of the design as a decomposition of modules, also referred to as design by levels of abstraction. Figure 17 illustrates the structure of a software system designed using the RDM. The rectangles represent modules, and their relations are intended to represent modular decomposition. Within a module, the circles represent programs, and their relations are intended to represent stepwise refinement, part of detailed design, discussed below. General responsibilities of the modules are decided upon, and the modules are arranged in a hierarchy. The hierarchical relationship between two modules is expressed in a "type definition" in the upper module which references the lower one. (See below.) General design involves naming each module (assumed here to be done in block 1.2.2.2 (line 4), of Figure 16, although it possibly should be done at block 1.2 or 1.2.2.2.1) and providing data for the four left-most blocks of line 6 in Figure 16. These blocks generally describe the data structures of the module and list its programs (operations). In addition, some of the text blocks discussed earlier are filled in as necessary for the lower-level modules. Presumably, at least initial drafts of at least some of these exist for the top-level module (the system) at the start of general design.

A detailed design phase involves definition of the programs of each module, those listed during general design, and supplying data for the remaining text blocks. Detailed design involves supplying data for the blocks within the outlined area starting at block 4 on line 5 of Figure 16. A program may be refined by including lower-level programs within it which are referenced by the higher-level program. The lower outlined area starting at block 4.4.1 represents a refinement of a portion of the program at block 4. Such a process is referrred to as program refinement or stepwise refinement.

Although it is appealing to think of a general design process followed, in time, by a detailed design process, Honeywell's current thinking is that such a situation is not possible in all cases. Detailed design of the upper-level modules may influence general design of lower-level ones, so that an individual designer may need to alternate activities between general design and detailed design. The activities and products of general design and detailed design are discussed further in the next two sections.

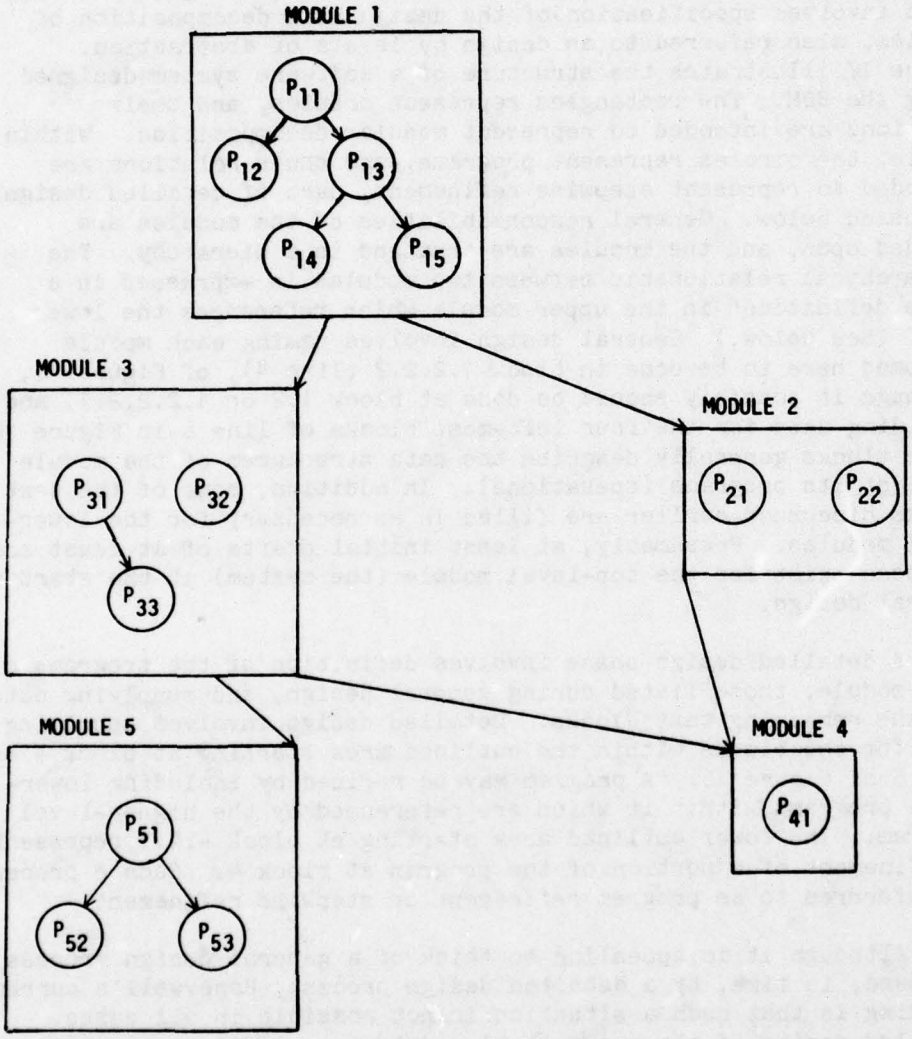


Figure 17. Honeywell Modular Decomposition Vs. Stepwise Refinement

GENERAL DESIGN PHASE

As modular decomposition is performed, new modules are defined. Figure 18 includes the blocks of a module which are primarily pertinent to general design. Several of the blocks in line 4 are of initial importance. The module must be named (block 1.2.2.2) and a functional description of the module provided for block 1.2.1.1. General and possibly detailed design for higher-level modules should provide guidance. The "usage-information" block for a new module needs to be filled in at least in parallel with detailed design of higher-level users of the new module. The "acceptance-criteria" block for the new module can be filled out as resource allocations between modules are performed. The information is needed before implementation of the new module and possibly by the start of detailed design for the new module. Blocks 1.2.2.1 and 1.2.2.3 are possibly more pertinent to detailed design.

Program Declarations

The four left-most blocks on line 6 relate to global declarations for the module. These blocks deal primarily with the data of the module since the details of its operations are the principal subject of detailed design. However, block (1.2.2.2.)3.4 declares the global operations of the module by listing the names (only) of the top-level programs of the module (those whose design will later be detailed at blocks 4, (1.2.2.2.)5, (1.2.2.2.)6, etc.). The scope of an operation (program) is not explicitly declared. An operation is intended to be referenced by higher-level modules. It is unclear if an operation is intended to be able to reference other operations within the same module.

Data Declarations

The first two blocks of line 6 are the principal ones related to data during general design (data local to the programs of the module are determined during detailed design). The global data of the module are declared in the "data" block. The "scope" of the data is explicitly declared. The apparent intent is that data in this block can be declared global to the module but not available to other modules (private) or it can also be global outside the module. Scope also allows specification of whether a data object is fixed (constant) or variable.

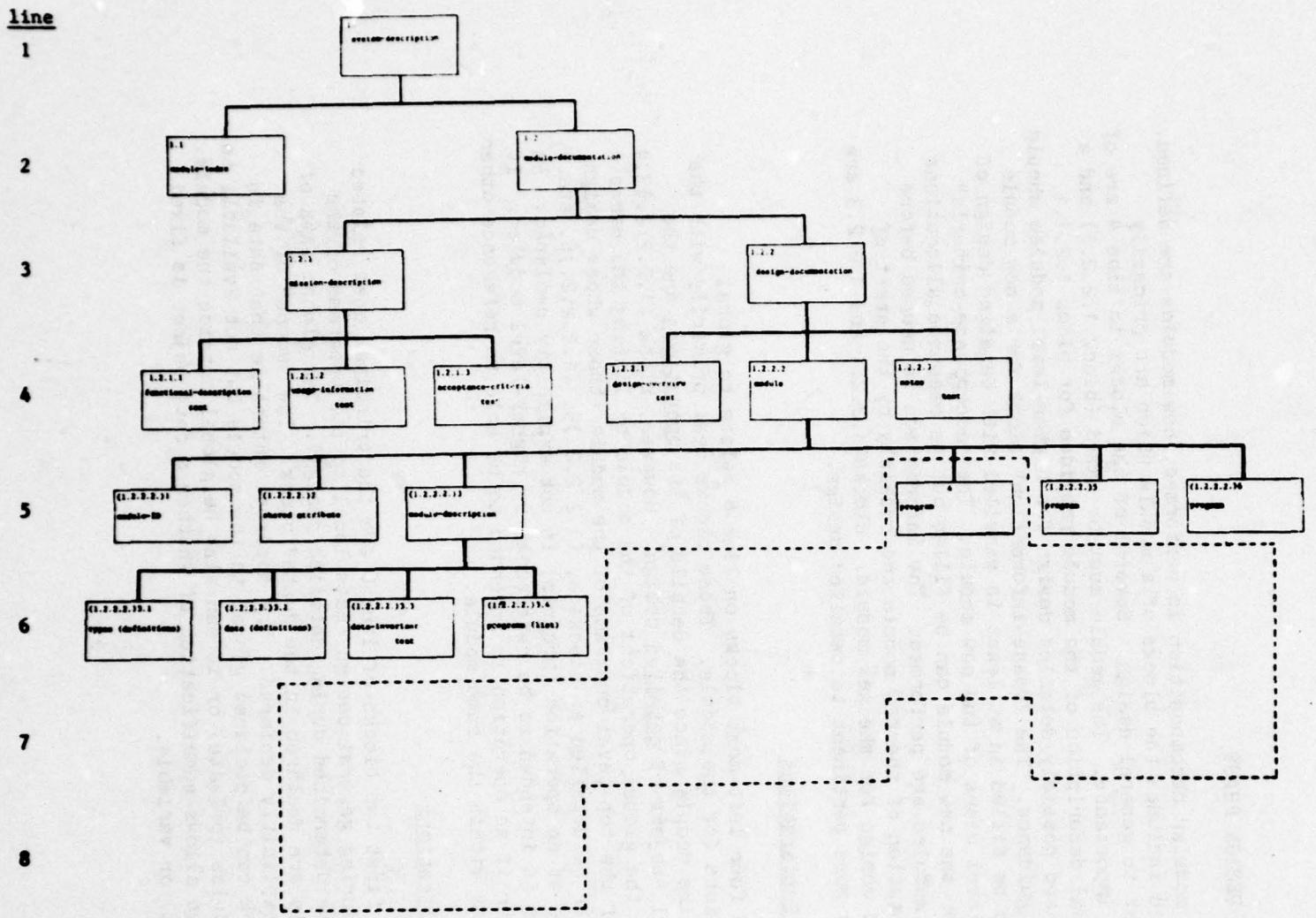


Figure 18. General Design

Type Declarations

Data declarations also indicate the type (or class) of a data object. The type of a data object not only determines the kind of information it may contain but also the kinds of operations that may be performed on the object. The PDL portion of the BNF definition of the syntax of the RDM provides various primitive data types such as integer, boolean, char, etc. Also provided are various facilities for constructing additional types. Declarations of such constructed types may include data components as well as operations available upon them. If the data declaration of a data object refers to a constructed type, then a type declaration for that constructed type must appear in the "types" block of line 6 for the same module, except in the case of arrays and sets, discussed next.

The PDL of the RDM includes type declarations for several common constructed types: arrays and sets. If L is declared to be of type array and the type of its elements is also declared, then various data components and operations become defined for the array L, some of which are shown in Figure 19. Some of the same kinds of data components and operations are available for sets, the principal difference between the two being that the elements of a set have no ordering.

The RDM allows types to be constructed which are structures of juxtaposed (or concatenated), simpler types, referred to as a Cartesian product of types. If date is declared to be a type consisting of the product of the types month, day, and year (which must also be declared as types), then a number of data objects could be declared to be of type date and each would have month, day, and year components. A type can be constructed which may be any one of a declared list of types, referred to as a union of types. If type number is declared to be the union of type integer or type real, assuming real were defined to exclude integer, then a data object declared to be of type number could contain either a real or an integer. The sequences of data objects or types of WELLMADE are not included in the RDM since arrays (present in both) provide adequate facilities.

If a single "stack", for instance, is needed in a system and is not referenced outside the module in which it is defined, it should be possible to define it in the "data" block of the module. If several different stacks with essentially the same kinds of characteristics are needed, all in one module, "stack" might be declared a type in the "types" block and the several stacks declared to be of type "stack" in the "data" block or as variables local to programs of the module.

If L is an array
and Ei is an expression such as A+B
and X is an identifier such as A,

then L.hib gives the value of the index to the top of the array (high bound).

L.lob gives the value of the index to the bottom of the array (low bound).

L.dom gives the domain of the index (L.dom = L.hib - L.lob + 1).

L.high gives the value of the element at the top of the array.

L.low gives the value of the element at the bottom of the array.

L.(E1) gives the value of the element whose index is equal to E1.

L:shift(E1) causes a shift in the index values used to reference elements in the array without changing the size or contents of the array.

L:swap(E1, E2) swaps the contents of the elements of the array whose index values are E1 and E2.

L:loext(X) adds an element to the bottom of the array whose value is that of X, if X is of the same type as the elements of the array.

L:hiext(X) adds an element to the top of the array whose value is that of X, if X is of the same type as the elements of the array.

L:lorem removes the bottom element of the array (and reduces L.dom by 1).

L:hirem removes the top element of the array (and reduces L.dom by 1).

Figure 19. Some Array Variables and Operations

Levels of Abstraction

If the concept of "stack" is to be treated as a facility so that one or more stacks are to be referenced by more than one module, then "stack" is treated as a so-called abstract data type. (Lind76 is a particularly good discussion of abstract data types and the advantages of their use). One lower-level module (named "stack") would provide the "stack" facility. That module would further define the data components of stacks and its programs would implement stack operations. The higher-level modules which use stacks would declare "stack" (including its data components and operations) as a type in the "types" block. Some portion of this type declaration, such as that used to declare the type of data to be stored in an object of type "stack", might be declared to be of type abstract, meaning that it will be declared more specifically elsewhere. In the "data" block of the same module, a particular data object could be declared to be of type "stack", the elements of which, rather than of type abstract, might be declared of type integer (or some other primitive or constructed type).

Abstract data types allow modular decomposition by levels of abstraction. When "stack" is treated as an abstract data type, the module which implements the stacks is at a lower level of abstraction than those which use them. Figure 20 briefly sketches some of the components which might appear in a lower-level module called stack which implements a stack facility and a higher-level module called upper which uses the stack facility. The example is based on material supplied by Boyd.

When it is felt that general design of a module is complete (except for possible effects on it of detailed design of higher-level modules), it may be possible to add more information to some of the text blocks of line 4. Detailed design for the present module might start at this point or be postponed pending further definition of higher-level modules.

DETAILED DESIGN PHASE

As noted earlier, detailed design involves the detailed definition of the programs of each module. Detailed design of some of the higher-level modules may need to be done before general design of some of the lower-level modules can be completed. In addition, some of the remaining text blocks need to be completed.

Figure 21 includes the areas pertinent during detailed design. Although most of the blocks are shown for reference purposes, the

```
Module upper

   types

      type stack : (type element : abstract;
                    maxsize : integer;
                    stacksize : integer;
                    error : boolean;
                    push : (stack■error)fun (in : e : element);
                    pop : (stack■error)fun ;
                    top : (element■error)fun;
                    empty : boolean fun;
                    full : boolean fun;
                    init : stack fun (in : maxsize))

   data

      A : stack
          A.element : integer

   programs

      stackint

   program stackint :

      variables
         x : integer
         :
         :

      text [program logic]

         A:init (in : 10)   [initializes the stack A with ten elements]
            :
            :
         A:push (in : 29)   [puts integer 29 on top of stack A or sets
            :                A.error true]
            :
         X := A.top         [reads element at top of A or sets A.error true]
            :
            :
         A:pop              [removes and throws away element at top of
            :                A or sets A.error true]
            :

end upper
```

Figure 20. Honeywell Abstract Data Type Example

```
Module stack

   data

      element : abstract array
      maxsize : integer
      stacksize : integer
      error : boolean

   programs

      push
      pop
      top
      empty
      full
      init

   program init :

      [Some aspects of the intended linkages between the two modules
      are unclear, and some may not yet have been decided by Honeywell.
      The function init has an integer input parameter, maxsize, and
      a result parameter which specifies an object of type stack. The
      parameters are said to be passed by declaring them, in the stack
      module, as variables of the module or of a program within the
      module. Presumably the parameters should also be associated
      with the function either under programs or program but the
      current BNF does not allow it. The linkages for the resulting
      stack are further complicated by the necessity to associate
      the data declarations for the upper module with the call to
      init. At any rate, the intent here is for the stack module to
      create the object A with space to store 10 integers plus its
      three parameters: maxsize, stacksize, and error. For A,
      maxsize is set to 10, stacksize to 0, and error to FALSE.]
      .
      .
      .

end stack
```

Figure 20. Honeywell Abstract Data Type Example (Concluded)

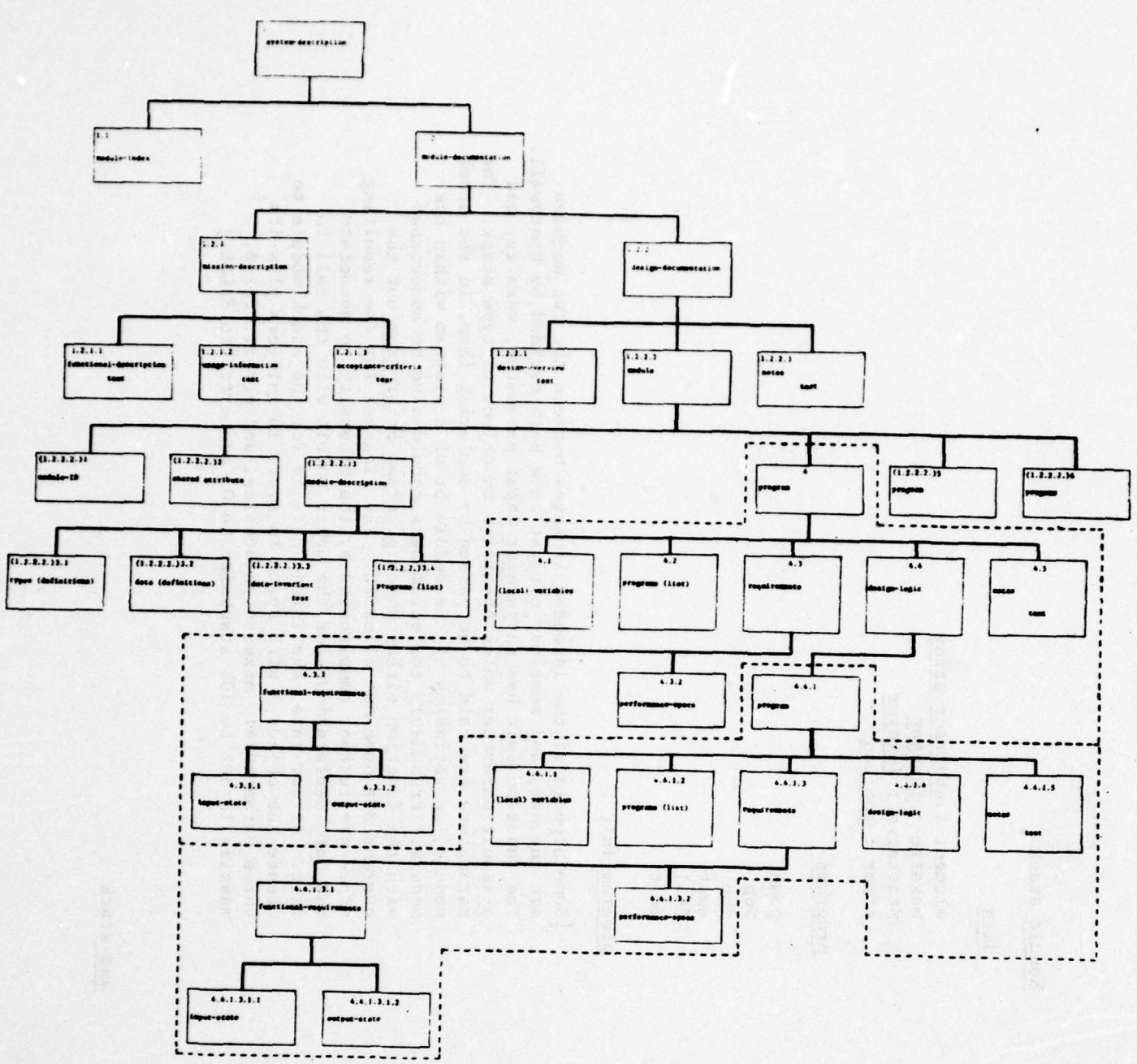


Figure 21. Detailed Design

ones of principal importance are the blocks within the two outlined areas and those on line 4, particularly those near the right end. At the start of detailed design of a module, the three left-most blocks should be complete or at least the information to complete them should be available. If the resource restrictions for the modules are not relatively clearly reflected in block 1.2.1.3 at this time for some reason, then alternative design strategies should be considered rather carefully to determine if any of them appear to require significantly greater amounts of one or more resources than others. Assuming that at least trend information is available as to which resources are or are apt to be most critical, design alternatives requiring the lesser amounts of these resources can be favored. Alternatives considered and reasons for choosing one as well as such things as assumptions made regarding implementation which relate to the whole module can be documented in the "notes" block of line 4. When a design alternative has been chosen and the design relatively well established, the "design-overview" block can be completed.

The top-level programs of the module which correspond to its operations were listed in block (1.2.2.2.)3.4 of line 6 during general design. The parameters used by these programs should be given in the "usage-information" block of line 4. The programs are now to be defined by supplying information for blocks such as those within the upper outlined area starting with block 4 of Figure 21. If the module contains more than one top-level program, then other "outlined" areas are assumed to start at blocks (1.2.2.2.)5, (1.2.2.2.)6, etc.

Program Refinement

The other kind of hierarchical decomposition mentioned earlier is that of program refinement. Stepwise refinement of a program within a module can be accomplished so as to delay lower-level decisions to points lower in the program hierarchy. Program blocks internal to other blocks are permitted by the RDM to the extent desired. The lower outlined area starting at block 4.4.1 represents a program block within the upper program. Each program block may contain zero or more other program blocks and each contains the same type of information, regardless of the level in the hierarchy, as indicated by the shape and contents of the two outlined areas in the figure.

The upper outlined area starting at block 4 will be used as a basis to discuss the manner of defining programs. Block 4 contains the program name. Block 4.5 can be used for notes pertaining to the design or implementation of the program. Block 4.2 is used to name

any lower-level program blocks contained within the program, such as one in the lower outlined area starting at block 4.4.1. Such lower-level programs might be expanded in-line by a macro processor or called as subprograms when implemented.

Blocks 4.3 and 4.3.1 are linkage blocks. Block 4.3.2 can be used for a text description of the performance requirements (specifications) of the program. The RDM, like most methodologies, currently contains no guidelines for assuring that the program to be defined will meet the performance requirements. The remaining blocks ("variables", "input-state", "output-state", and "design-logic") are the principal ones used for definition of the program.

Program Variables

Block 4.1 contains declarations of variables local to the program and possibly declarations relating to variables of its ancestor programs and the data of the module. In general, the rules which apply to the "data" block (1.2.2.2.3.2), as discussed under General Design, including scope rules, apply to block 4.1 also. If some portions of a local data structure are to be declared in a lower-level program block, those portions would be declared of type abstract in the declaration for the current program.

Program Requirements

Blocks 4.3.1.1 and 4.3.1.2 are used to make assertions regarding the state of the module at entry to the program (input state) and at exit from the program (output state). These states are usually described in terms of values of the state variables of the module. (If the module is viewed as implementing an abstract data type, then the state variables correspond to the data items of the data type, i.e., not the operations.) Output states may be contingent on input states. The only state variables that need be considered are those which relate to the actions of the program, all others being assumed to be unaltered by execution of the program.

Program Design Logic - Constructive Approach

The last major step of detailed program definition is to supply a description of the "design-logic" for the program for block 4.4. This description is to be constructed (using techniques referred to as Dijkstra's constructive approach) so that the design-logic accomplishes its intended functions and satisfies the input and output assertions. The approach is amenable to formal proof-of-correctness but even used in the absence of formal proofs, it is

expected to reduce design errors since proof considerations are to be a part of the construction process.

The general rationale of the constructive approach is as follows. The "output-state" assertions specify the set of acceptable states of the module at exit from the program. Since the purpose of the program is to surrender control with the module in one of these acceptable states, assume that the state of the module at exit is described by the "output-state" assertions. The design logic is to describe a program which will transform the state of the module from one of the expected "input-states" to one of the acceptable "output-states". The design logic is to be constructed using a set of well-defined building blocks or constructs which have known state transformation characteristics and whose effects are localized (no branches, for instance) so that the transformation produced by a set of steps can be determined by looking at the transformation produced by each step. More particularly, knowing the transformation effects of each construct, one can work backwards, stepwise, from the assumed, final "output-state" for each path through the design logic. At each step, one can determine the set of input states which, transformed by the construct currently being examined, will result in the set of output states already determined. In this manner, one can determine the set of all initial input states (at entry to the program) which, when acted upon by the program, will result in the set of output states assumed at exit from the program. This set of input states is thus derived from the output set based on the transformation characteristics of the design logic specified for the program. If the expected set of input states described in block 4.3.1.1 is a subset of the derived set, then it has been demonstrated that the design logic will transform the module from an expected input state to an acceptable output state. In deriving a set of input states from the output states, it will be necessary to demonstrate that any loops in the program terminate successfully. Specifically, the "constructive approach" is intended to imply that programs are constructed by allowing the set of output states to suggest the program, by applying the constructs while considering their transformational characteristics so that the set of all <u>possible</u> input states includes the set of <u>expected</u> input states.

Note: In the terminology of Boyd and others, the relationships, such as $x < 5$, used to describe the expected input states and the acceptable output states are "predicates". Thus, when a program or a construct is viewed as a transformer, it is called a "predicate transformer". An output state relationship is also referred to as the "post-condition" and an input state relationship as a "pre-condition". The derived input state relationship based on a known

"predicate-transformer" and a known "post-condition" is referred to as the "weakest pre-condition", which encompasses the largest set of input states with the appropriate characteristics. End of Note.

Note that proof-of-correctness, as described above and as applied in other methodologies, does not prove that a program "does what it is supposed to do", in a general sense. The program is viewed as a transformer to transform the state of the module from any of the expected input states to an acceptable output state, and that is the operational definition of "what the program is supposed to do". It seems unclear that it is possible in all cases to draw the input assertions and output assertions so finely that it can be proved that the proper output state results for each expected input state, other than by examining all states individually which is not the intent.

Constructs and Transformational Characteristics

The last aspect of the constructive approach to examine is that of the transformational characteristics of the individual constructs used in specifying the design logic for a program. These constructs include several of a functional form and the so-called P-Notation constructs of Dijkstra. The constructs of a functional form are generally separately constructed, subject to separate proofs, and their transformational characteristics are specifically stated. The array and set functions provided by the PDL would presumably be implemented and proved once. The operations of lower-level abstract machines are their programs which are constructed and proved like other programs. Lower-level program blocks are handled separately in the same way. The linkages between these abstractions must also be known to be correct.

The P-Notation constructs are described in Figure 22. Each of these has transformational effects which seem intuitively clear except for the last. In each case we need to describe the set of input states which must exist if the construct as a transformer is to produce a given set of output states. In the case of the null or skip construct, the derived set of input states is identical with the output set since skip performs no state changes. For abort, there is no input state which will produce any output state since abort never reaches output (the input set is the false or null set).

For the assignment construct, the input set is derived by substituting the assigned expression into the output assertion. If the output assertion is $y < 10$ and the assignment is $y := 2x$, then the input assertion is $2x < 10$ or $x < 5$. The derived input set for a sequence (concatentation) of two constructs (S1; S2) is determined

NAME	GENERAL FORM	EXAMPLE
1. NULL	skip	skip

Discussion: No action occurs and the next statement in sequence is executed. Not intended to be used by itself but in conjunction with the conditional statement types, selection and iteration, discussed below.

2. Abort	abort	abort

Discussion: No further execution takes place in the abstract machine in which it is encountered. Not intended to be used by itself but in conjunction with the conditional statement types, selection and iteration, discussed below.

3. Assignment	V1, V2, ..., Vn := E1, E2, ..., En	x, y := x+5, 2x

Discussion: The value of the variable Vi is replaced by the value of the corresponding expression Ei. Shown is the multiple or simultaneous assignment statement. Dijkstra's intent (and presumably Honeywell's) is that in the case of the multiple assignment, all assignments take place simultaneously, in multiple processors, for instance. The values of all variables appearing in expressions on the right-hand side are the old values as of the start of execution of the statement (the old values are all captured at the start of execution). If the multiple assignment is executed by a single processor, then no sequence of the individual assignments should be assumed since sequence of execution has no effect on the results. In the example above, if it is intended that the new value of y be based upon x+5, the new value of x, then two single assignment statements must be used.

Figure 22. P-Notation Constructs

NAME	GENERAL FORM	EXAMPLE
4. Concatenation	S1; S2	x := x+5; y := 2x

Discussion: First execute S1, then execute S2. A sequence of statements separated by semicolons is executed.

5. Selection	if G1 -> S1 ■ G2 -> S2 . . . ■ Gn -> Sn fi	if x < 0 -> abort ■ (0 ≤ x) and (x ≤ 1) -> y := sqrt (x) ■ (1 ≤ x) and (x < 1024) -> y := x * sqrt (x) fi

Discussion: The Gi are Boolean expressions called guarding heads or guards. If none of the Gi evaluates true, then the selection statement is equivalent to abort (in the example, this will occur if $x \geq 1024$). If one or more of the guards is true, then the statement Si (which may be a concatenation of statements), associated with a single arbitrarily selected, true guard is executed once. In the example, if x = 1, then both the second and third guards are true. But in this case, the associated statements, S2 and S3, both produce a value of 1 for y, so that it is immaterial which one is executed, except for efficiency. This type of situation in which several guards are true but the statement associated with each produces the same result is one mentioned in particular by Dijkstra. It is an unneeded distraction to have to ponder which set the boundary value does or should be placed in, in such cases, and it is really better to specify non-determinacy. The intent is less clear when two or more guards are non-disjoint but their associated statements produce different results. Honeywell has also mentioned the option of restricting such statements to the determinate case in which the sequence of appearance of the true guards affects the selection of the associated statement to be executed. (The first guard and its associated statement illustrate explicit use of the abort statement. In this example, the effect would be the same if they were omitted.).

Figure 22. P-Notation Constructs (Continued)

NAME	GENERAL FORM	EXAMPLE
6. Iteration	do G1 -> S1 ■ G2 -> S2 . . . ■ Gn -> Sn od	do A(1) > A(2) -> A:swap (1,2) ■ A(2) > A(3) -> A:swap (2,3) ■ A(3) > A(4) -> A:swap (3,4) od

Discussion: If all the guards are false initially, the iteration statement is equivalent to skip; otherwise, statements associated with true guards are repeatedly executed until all guards are false at which point skip occurs. As in the case of the selection statement, all guards are evaluated and the statement(s) associated with an arbitrarily selected, true guard are executed; the guards are then re-evaluated to determine if the process is ended or if another true guard is to be arbitrarily selected. The assertion is made that there exists a class of tasks which can be performed by the execution of n steps, possibly repeatedly, and that the end result is the same regardless of the order of execution of the steps. For such tasks, specifying an execution sequence is an over-specification and an undesirable activity. If sequence of execution of the steps is of importance, then the construct must be made determinate or used only in a determinate manner. Honeywell's intent is not clear in its documentation. The example of the construct given above, which is clearly not generally applicable, does an in-place sort of the values in a small array, A, by repeated comparison of adjacent values and swapping them if the one in the upper position does not have the higher value. When all adjacent pairs have the proper relationship, the process is ended. (A more practical example of in-place sorting is the bubble sort used in Figure 23 to illustrate the Honeywell methodology.)

Figure 22. P-Notation Constructs (Concluded)

as stated previously, by examining each separately. The input set for S2 is determined based solely on S2 as a separate transformer. The derived input set for S2 is the output set for S1 so that the input set for S1 can be derived by considering only S1 as a transformer. The derived input set for S1 determined in that manner is also the derived input set for S1; S2.

The derived input set for the selection or *if* construct is determined separately for each possible path and the results expressed as the ORed combination. The effect of the guards must also be considered. If the construct is:

```
if G < 1000 -> T := .1*G.
 ■ G ≥ 1000 -> T := .2*G
fi
```

and the output assertion is T < 500, then the derived input set for the first assignment is .1*G < 500 or G < 5000. But since the first assignment has no effect unless G < 1000, then the derived input set for the first path is G < 1000. The derived input set for the second assignment is .2*G < 500 or G < 2500. Combining this effect with that of the guard gives a derived input set for the second path of 1000 ≤ G < 2500. The combined derived input set is G < 1000 *or* 1000 ≤ G < 2500 which is equivalent to G < 2500.

In the case of the last construct, iteration or the *do* loop, the formal definition of the transformation is not intuitively useful for constructing programs. Therefore, an invariance theorem is used to suggest the proper method for loop construction. The output assertions specify the required relationships between the state variables of the module at exit from the loop construct. These output assertions are separated into two sets of assertions such that the simultaneous satisfaction of both sets is equivalent to the satisfaction of the output assertions. One set of assertions represents an invariant which is to be satisfied by the state variable values at every point within the loop except while the guarded statement(s) associated with one true guard are being executed. In particular the invariant must be satisfied (possibly vacuously) prior to the first pass through the loop and after the last pass. The other set of assertions that, together with the invariant, make up the output assertions, represents the termination condition for the loop. The termination assertions are false everywhere within the loop except following the last pass. Thus, after the last pass the two components of the output assertions, the invariant and the termination condition, will both be satisfied and, therefore, the output assertions are satisfied.

The guard(s) of the loop perform the termination tests. When all guards are false, the termination conditions have been satisfied and exit from the loop will occur. The guarded statements characterize or perform the function of the loop. The guarded statements must systematically step toward termination as well as maintaining the truth of the invariant everywhere within the loop except while they are actually being executed. Prior to entry to the loop, the state variables must be initialized, if necessary, so that the invariant is true at the beginning of the loop and the termination condition is false (except for those cases in which the output assertions are already true at the beginning of the loop). If all of the above conditions are satisfied then the loop will correctly perform its function and will terminate properly. The example given below may help to clarify the application of the invariance theorem to proper loop construction.

TOOLS

The RDM has no tools, as yet. Tools for syntax checking to support verification of general and detailed design specifications are being investigated. It is hoped that those existing WELLMADE tools which support general design can be adapted for use in the RDM. (Such tools would need to be modified to account for differences in such things as design structure and document formats.) Consequently, it is pertinent to discuss briefly the WELLMADE tools. However, it seems possible that the greater complexity of the RDM than that of WELLMADE may further complicate the conversion of WELLMADE tools.

The earlier versions of WELLMADE viewed a system as a relatively pure hierarchy of modules. The top-most module would represent a relatively abstract view of the entire system. Proceeding downward, each level would utilize the modules of the next lower level of the hierarchy. Each level was to represent an increasingly detailed set of capabilities. Although it was noted that a module and its occurrence in the tree structure might differ from the physical manifestation (design structure different from implementation structure), the discussion below depends rather heavily on the concept of a relatively pure hierarchy of homogeneous elements (modules). It appears that the WELLMADE "modules" were primarily programs - procedures, subroutines, and coroutines.

Table I outlines a design description for one module, based on Boyd76. The description is divided into two categories, each of which is subdivided into various subjects. Multics is used to maintain the WELLMADE documentation. Material for each subject for

Table I

Outline of Earlier WELLMADE Module Description

Mission Description

- Functional Description
- Usage Information
- Acceptance Criteria

Design Documentation

- Design Overview
- *Variables
- *Input States
- *Output States
- Function List
- *Program Description and Proof
- Notes on Design
- Notes to Implementor
- Test Plan
- Implementation Plan

* Most significant sections, which contain the formalized design description.

each module (a documentation element) is separately identified for purposes of creation, updating, retrieval, etc. An official directory is used to reference elements of the documentation that are complete and accepted. A scratch directory references documentation elements while in preparation. For dissemination of information, any project member can read anything referenced by either directory. Only those responsible for a documentation element may update it through the scratch directory, and only the librarian (and project manager) can update through the official directory. Progress is assessed based on the status of the documentation. One can determine if a documentation element has not been started, is in progress, or has been accepted. The status of the documentation elements for a given module gives an indication of the degree of completion of work on the module. Modification dates for documentation elements indicate when various stages of a module were completed or last worked on.

The above facilities are generally provided by Multics and general support packages. A specialized support package (doca) is used to output documents, providing standardized formats, page numbering, and tables of contents. The directories identify the documentation elements for different subjects and modules, and, for each module, the so-called Function List identifies "submodules". Documents can be prepared for any category and/or subject(s), for the system or any subtree, starting with any module and proceeding downward any specified number of levels. Another specialized support package (P-Notate) simplifies entry and formatting of material using P-Notation constructs. V-Notate is used to create a data base of variables and information about their use.

Not shown in Boyd's version of a WELLMADE module description (Table I), is a third category of information relating to the implementation of a module, including subjects such as a code listing, test plans, and test report. To the extent that there is not a one-to-one correspondence between design and implementation, the hierarchy is less pure.

Recently WELLMADE has added a fourth category for planning and tracking whose subjects include a development plan and tracking of actual progress, as well as design and implementation inspection reports. WELLMADE is also being embodied in a handbook of procedures which include various elements similar to those of the Air Force acquisition environment, such as several levels of reviews and various types of boards.

EXAMPLE

Figure 23 is a short example of use of the RDM. The numbers at the left of the figure are not part of the methodology but, as before, correspond to the block numbers in Figure 24. Figure 24 is similar to Figure 16 but with the blocks highlighted that correspond to the design documentation segments which appear in the example.

The example was provided by Don Boyd and Stan Vestal of Honeywell and is the only one available as yet of use of the RDM. (Several partial examples of WELLMADE are available in non-public documentation plus one in BOYD76. All complete examples of WELLMADE are for currently-operational software and are considered proprietary by Honeywell.)

The example represents only part of one module and, therefore, illustrates modular decomposition (the use of an abstract data type) only very superficially: the module can be considered a lower-level one providing a facility for use by higher-level ones which are not shown. The example does illustrate refinement of a portion of a top-level program by a second-level program.

The RDM specifies the use of braces (curly brackets) to set off comments or unrestricted blocks of text. Such material is not intended for processing by computer other than for storage and printing of it. Because of type-font differences, braces have been shown in Figure 23 as (square) brackets. Note that the input and output requirements in both programs are, therefore, treated as text. Although they are important to the designer, they are only for human use at present.

The module is named BUBBLE and is intended to perform an ascending bubble sort of an array of integers, L (an in-place rearrangement of the contents of L so that $L.j \leq L.j+1$ for $L.\underline{lob} \leq j < L.\underline{hib}$). (For notation, see Figure 19.) The first page of the example contains global declarations for the data and top-level programs of the module as well as the top-level program, sort, itself. The second page contains the second-level program, insert, which is contained within sort, and also sketches the existence of the two other top-level programs, load and print.

In the first page of the example, L is declared to be an array of integers at the "data" block, (1.2.2.2.)3.2. No scope is shown for L so presumably some unspecified default applies. L is used by all the programs of the module so it has to be global to the module (has to be included at the "data" block rather than as a program

variable). Since its contents are to be rearranged, if necessary, it must be a variable rather than a constant.

The "programs" block, (1.2.2.2.)3.4, lists the three top-level programs of the module: sort, load, and print. Sort is the only one specified in the example and is discussed below. All we know about the load and print programs is contained in the comments beside them at the "programs" block: load initializes L and print prints L. With regard to further clarification of what the scope of L should be, these programs provide operations for use by higher-level modules to put unsorted data into L and remove sorted data via a printer. If no higher-level module needs any other access to L than these, then L can be private to BUBBLE. The only other sensible kind of access to L within the known constraints of BUBBLE is that one or more higher-level modules may need to access the sorted L for further processing of its contents. In this case, L must be global outside the module. (It is not suggested that the designer of a module normally has to guess the uses to which it will be put. In a given system, enough should be known about the higher-level users of BUBBLE to determine what the scope of L should be. The purpose of the discussion is to illustrate further the scope concept.)

The rationale of the program sort is simply to make one pass over the array L, starting at its bottom (L.lob), and to determine if it is already in ascending sequence, by comparing the contents of each location with that of the next lower location. If sort finds a location whose contents is less than that of the previous one, then it calls upon its lower-level program, insert, to re-establish the ascending order. Although the designer must have reasonable confidence that he can design a program called insert to perform its intended function, he is generally intended to ignore other aspects of insert while designing sort.

The program sort has one local variable, i, whose type is integer, to be used for indexing the locations of the array L. Since i is declared within sort, it is local to sort (and possibly to lower-level program blocks within it). Its scope is, therefore, private and, since it has to be used to access each location of L, variable rather than constant. The program insert is named at block 4.2 as an internal program block of the program sort.

Since L has been declared to be an array, the array data components and functions described in Figure 19 are provided by the PDL for referencing and manipulating the array. In essence, array can be treated as though it were the name of an abstract data type, but one which is built-in to the PDL and methodology so that an

```
1.2.2.2          Module BUBBLE:
(1.2.2.2)3.2       data
                     L: integer array                    [list to be sorted ]
(1.2.2.2)3.4       programs
                     sort                                [a bubble sort which rearranges L into
                                                          ascending order of integers]
                     load                                [a program which initializes L ]
                     print                               [a program which prints L ]
        4          program sort:
        4.1          variables
                       i: integer   pv                   [index of the list L ]
        4.2          programs
                       insert                            [a routine to insert an integer into a sorted
                                                          list]
        4.3          requirements
        4.3.1.1        input [L.dom ≥ 1]
        4.3.1.2        output [L.(j) ≤ L.(j+1) for
                                 L.lob ≤ j < L.hib ]
        4.3.2          performance [to be supplied]
        4.4          text [design logic]
                       i := L.lob;                       [ L.(j) ≤ L.(j+1) for L.lob ≤ j < i ≤ L.hib
                                                          is the invariant and i ≥ L.hib is the
                                                          termination condition]
                       do i<L.hib →
                           if L.(i) ≤ L.(i+1)→skip
                           ▮ L.(i) > L.(i+1)→insert
                           fi;                           [the invariant has been reestablished ]
                          i := i+1
                       od                                [the output requirement has been established]
```

Figure 23. Honeywell Example

Note: [ ] have replaced { }

```
                    begin
(1.2.2.2.)4.4.1       program insert :
        4.4.1.1         variables
                          j: integer    pv      [index for the list L]
                          i: integer    gc      [index the element to be inserted]
        4.4.1.3         requirements
        4.4.1.3.1.1       input [(L.(k) ≤ L.(k+1) for
                                 L.lob ≤ k < i and
                                 L.(i) > L.(i+1), for
                                 i > L.lob) or
                                (L.(i) > L.(i+1) for
                                 i=L.lob)]
        4.4.1.3.1.2       output [L.(k) ≤ L.(k+1) for
                                  L.lob ≤ k ≤ i]
        4.4.1.3.2         performance [to be supplied]
        4.4.1.4         text [design logic]
                          j := i+1;                 [L.(k) ≤ L.(k+1) for j ≤ k ≤ i is the invariant and
                                                     j ≤ L.lob or L.(j) ≥ L.(j-1) is the termination
                                                     condition]
                          do j > L.lob and
                               L.(j) < L.(j-1) →
                                     L:swap(j,j-1);
                                     j := j-1
                          od
                    end
1.2.2.2.5           program load:
                      :
                      .                                  Note:  [ ] have replaced { }
1.2.2.2.6           program print:
                      :
                      .
                  end BUBBLE
```

Figure 23. Honeywell Example (Concluded)

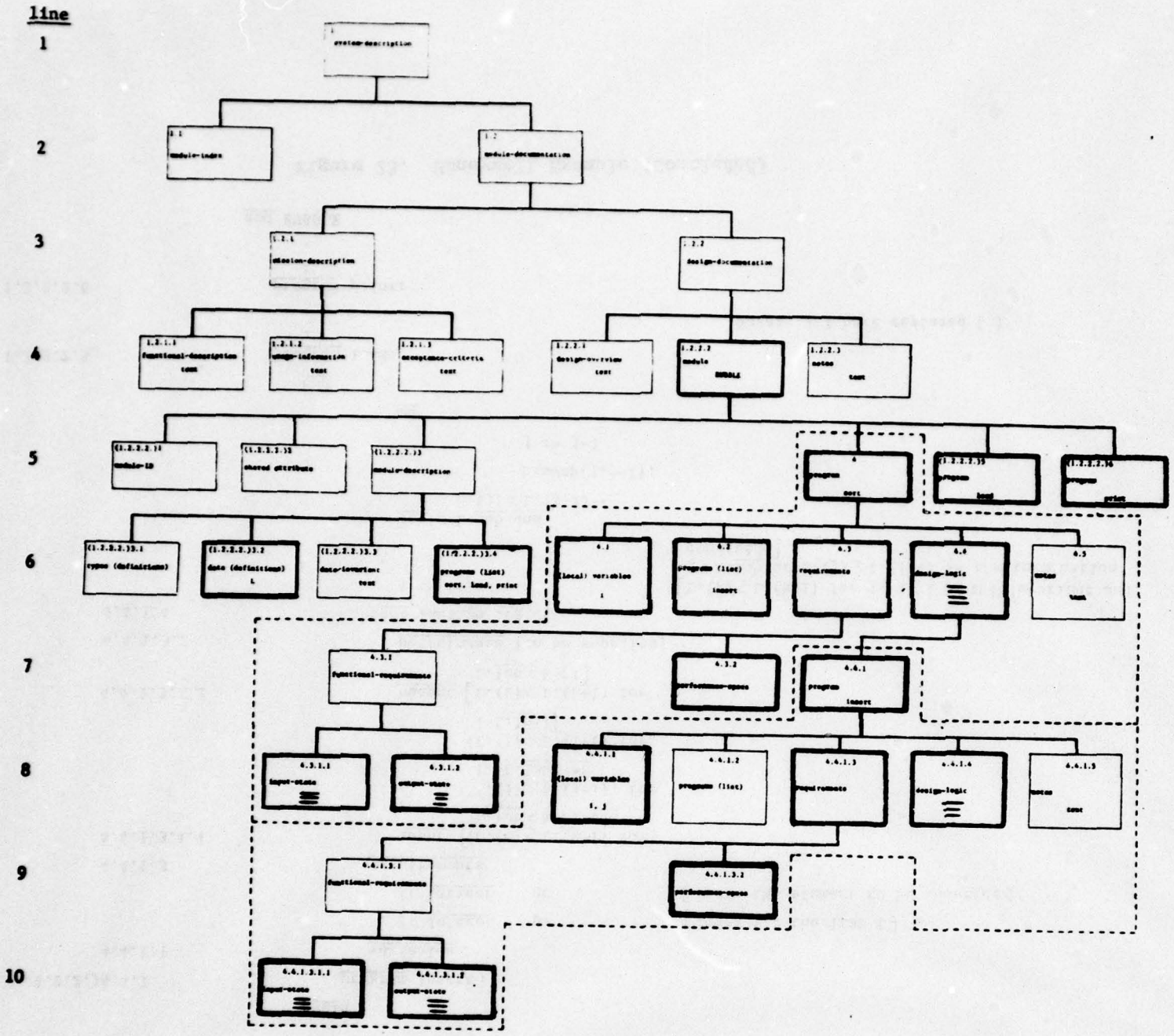


Figure 24. Components of Honeywell Example

array module need not be designed for each system it is to be part of. The data and operations of the array module are defined by the PDL rather than by a type declaration at the "types" block.

The "input requirements" (block 4.3.1.1) for sort specify the expected state of the module at the time sort is entered. The statement says that L is expected to consist of at least one element at entry to sort and it implies that sort is not expected to behave properly if L is empty but that if L consists of at least one element, sort is expected to perform its intended function and terminate properly. Notice here at least a weak example of the vagueness of English and the greater precision and conciseness of the "input requirements" statement. The verbal description of the rationale of sort several paragraphs back is phrased in terms of making comparisons between the contents of successive locations, but the more formal requirement is that sort must not only behave and terminate properly with two or more elements but also with only one. It is granted that the textual description and the formal requirement may be made to agree, either by extending the textual description by a moderate number of words and/or by changing the formal requirements to disallow the case of a single element. If a higher-level module can perform its intended function with only one element to be "sorted", then it is probably preferable to allow sort to handle the case of a single element, treating that case as a "lower-level decision". At any rate, it can be seen below that sort terminates properly when L consists of a single element.

The "output requirements" (block 4.3.1.2) for sort specify the required state of the module BUBBLE at exit from sort, assuming that the "input requirements" were satisfied at entry. The "output requirements" specify the ascending sequence of values in L at exit, that for every pair of successive locations in L, the lower location may not have a greater value than the higher location. It might be argued that the "output" assertion should be extended by "or L.dom = 1" since the assertion given does not seem to apply with a single element in L.

The "design-logic" (text) at block 4.4 contains the formal definition of the actions to be taken by sort. The design logic consists of an assignment construct to initialize the applicable state variable, of the program in this case, prior to a do loop. The do construct contains one guard (i < L.hib). A sequence of two constructs comprises the guarded statement associated with the single guard of the do construct. These two are: (1) an if construct, itself with two guards, each of which is associated with a single guarded statement, and (2) another assignment construct. The discussion which follows is primarily related to use of the

invariance theorem to construct a proper loop statement; in the process, the other statements are explained since they are all related to the loop.

In the output requirements assertion (block 4.3.1.2), j is a dummy variable (not a state variable) which can take on any value in the indicated range. The output assertion is a statement which must be satisfied by the state variables at exit from sort. The first comment to the right of the program logic specifies (1) the invariant condition which is to be true before, during (except while guarded statements are executed), and after the loop, and (2) the termination condition which is only true following the last pass. The variable j is again a dummy variable. It should be clear later that at termination, the program variable i will be equal to L.hib and not greater. If this value of i from the termination condition is substituted for i in the invariant, the result is an assertion about the values of the state variables at exit from the loop which is also the exit from the program. The resulting assertion is equivalent to the output requirements assertion, as noted in the earlier discussion of the invariance theorem. Constraining i to be less than or equal to L.hib in the invariant is not necessary at termination but for previous, intermediate passes through the loop.

The first assignment statement in the program logic sets the program variable i to the bottom element of the array. It also (trivially) establishes the termination condition false unless L.lob = L.hib (the array contains only one element). If L contains one element, then the output assertion is already true at the beginning of the loop and the do loop should perform no state changes, which will be found to be the case shortly.

The single guard in the do loop (i < L.hib) performs the test for termination. When that guard is false, the do loop will terminate. In particular, with one element in L, then i = L.hib, the guard is false, and the do loop terminates without performing any state changes since the guarded statements are never executed.

With more than one element in L, the guarded statements are executed at least once. The if statement indicates that if the contents of location i of L is less than or equal to that of location i+1, no action is taken by the if statement (skip is executed which corresponds to no action). However, if the lower location of L contains a greater value than the higher location, then insert, the lower-level program within sort, is invoked to restore the ascending order within the first i + 1 locations of L. In the case of this if statement, the two guards specify non-intersecting sets of states, but the union of both sets is the

universal set, so that one of the two guarded statements must be executed on every pass through the if statement and abort cannot occur. At the end of the if statement, the invariant is still true or is again true (reestablished, in the words of the comment).

After the if statement, the second assignment statement steps the index to L by 1 and also steps toward satisfaction of the termination condition. At this point, the guard is again tested to determine if another pass through the do loop is called for. The program variable i was originally set to L.lob. By definition, L.lob $\leq$ L.hib if L.dom $\geq$ 1. Therefore, since both the index bounds in L are integers and i is stepped by 1 for each complete pass through the do loop, then at exit, i = L.hib so that the simultaneous truth of the invariant and the termination condition at exit from the do loop is equivalent to the output assertion for the program, which has the same exit point as the do loop.

The number of complete passes made through the do loop is L.dom-1 and the guard is tested L.dom times. The array L has been sorted by virtue of the fact that at the end of the ith complete pass, the first i+1 locations of L contain values in ascending sequence (as the invariant states).

The program insert which is internal to sort is discussed next but in considerably less detail than sort. The program insert is invoked by sort when sort has found two consecutive locations containing descending values. When insert is invoked, the local variable i of sort points to location i in L, whose contents is greater than that of location i+1. The program insert works by moving the value in location i+1 toward the bottom of L, one location at a time, until the ascending order of the first i+1 locations of L has been reestablished. The original contents of location i+1 is swapped with the contents of each preceding location until it is no longer less than the contents of the preceding location or until the bottom of L is reached.

The program insert uses the local variable i of sort as a global constant. A variable, j, local to insert, is used to mark the current position in L of the value originally at position i+1. Consequently, j is initially set to i+1 and then decremented by 1 for each pass through the do loop.

The input requirements state that on entry to insert, the first i locations of L contain ascending values but the contents of location i+1 is less than that of location i, if i is greater than L.lob. If i = L.lob, then only the second of the two assertions is true. The variable k is a dummy variable. The output requirements

state that on exit from insert, the first i+1 locations of L are again in ascending sequence.

As noted, j is decremented from an initial value of i+1 toward a value of L.lob. The invariant for the do loop states that the portion of L between locations j and i+1, inclusive, is in ascending sequence throughout the loop. Termination occurs when the bottom of L is reached or when the value in location j of L (formerly in location i+1) is greater than or equal to the value in location j-1.

The program logic initializes j to i+1. Then the if statement is used to swap the contents of locations j and j-1 in L and to decrement j by 1, until the termination condition is true. At least one pass through the do loop is necessary since insert is not called unless the value in location i+1 needs to be moved down one or more locations in L.

SECTION V

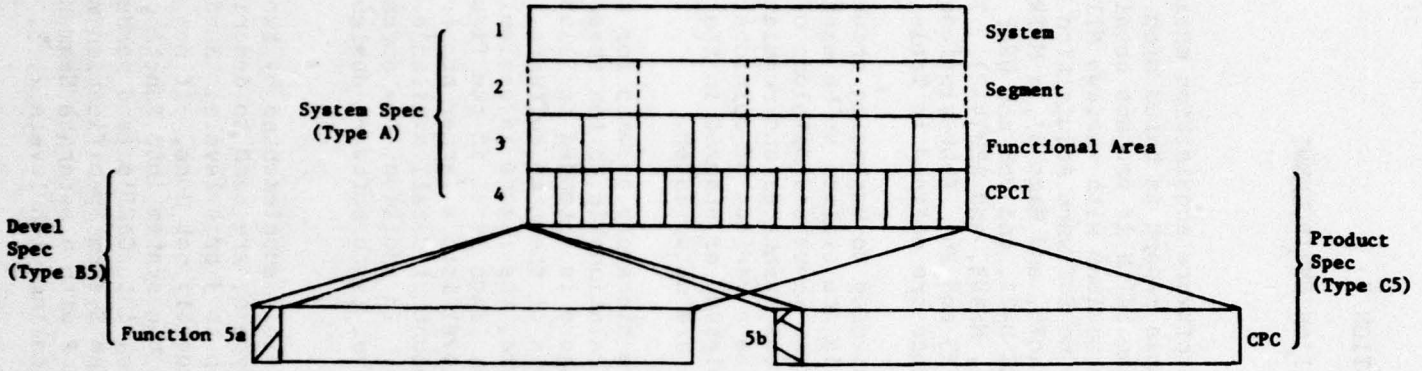
MODEL ACQUISITION ENVIRONMENT

The model of the Air Force software acquisition environment used for the work described in this report is based upon some prior experience, some experience on the SATIN IV project obtained concurrently with this work, discussions with various MITRE personnel, and study of some of the Software Acquisition Management Guidebooks (primarily Glor77, Scho76, and Haga75), a MITRE Working Paper by Bruce Feldmeyer of MITRE D-71, and DoD and USAF standards and regulations (primarily MS490, MS483, and AFR14-2). The assumed model is illustrated in Figures 25 and 26. Abbreviations (acronyms) used in this and following sections are defined in Table II.

The acquisition model is intended to emphasize principal software acquisition events and is not claimed to be exhaustive. Since various people have different views and opinions of the acquisition environment and various standards and regulations seem to disagree at times, there may be a variety of opinions as to events which should be added, deleted, or changed in the model. A few items not shown in Figure 26 are noted later.

Figure 25 shows the software hierarchy assumed for the present acquisition environment and its relationship to the three pertinent types of specifications. The figure is intended to illustrate five (or six) different representations of the same software. Since all levels represent the same software, the figure is basically a rectangle rather than a triangle. Each level in the figure is a subdivision of the next higher level into a larger number of smaller cells. In general, the total amount of detail available increases as the number of cells increases. In addition, the emphasis shifts from software requirements at level 5a to software design at level 5b.

A System Specification, possibly supplemented by two or more Segment Specifications, all of type A, are used to describe software (and hardware) requirements at up to 3 or 4 levels. Subdivision of the system into segments is frequently not done. If not done, then the System Specification divides the system into functional areas (usually) and each functional area into CPCI's (and hardware CI's). If segments are included, then the System Specification may primarily relate to levels 1 and 2 with a separate Segment Specification for each segment dealing with levels 2, 3, and 4.



CPCI = Computer Program Configuration Item

CPC = Computer Program Component

Figure 25. Assumed Acquisition Environment Software Hierarchy

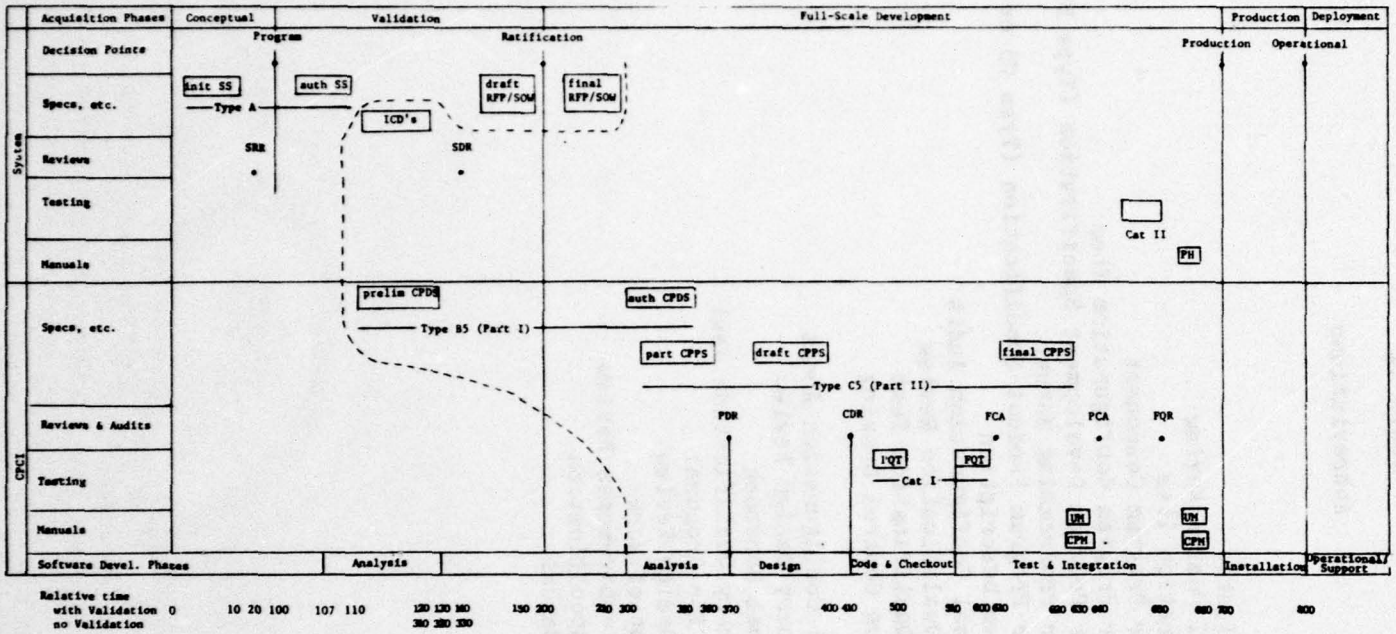


Figure 26. Assumed Major Acquisition Events Model

Table II

Abbreviations

auth	authenticated
CDR	Critical Design Review
CI	Configuration Item
CPC	Computer Program Component
CPCI	Computer Program Configuration Item
CPDS	Computer Program Development Specification (Type B5 or Part I)
CPM	Computer Programming Manual
CPPS	Computer Program Product Specification (Type C5 or Part II)
DID	Data Item Description
FCA	Functional Configuration Audit
FQR	Formal Qualification Review
FQT	Formal Qualification Test
ICD	Interface Control Drawing
init	initial
PCA	Physical Configuration Audit
PDR	Preliminary Design Review
PH	Positional Handbook
PQT	Preliminary Qualification Test
RFP	Request for Proposal
SDR	System Design Review
SOW	Statement of Work
SRR	System Requirements Review
SS	System Specification
UM	User's Manual

The two other types of specifications deal with levels 4 and 5. For each Configuration Item (software or hardware), a Development (or Part I) Specification and a Product (or Part II) Specification are required. For computer program CI's, the corresponding specifications are called the Computer Program Development (or Type B5) Specification and the Computer Program Product (or Type C5) Specification. A B5 specification subdivides a CPCI into functions and describes the requirements of the CPCI and of each function (level 5a in Figure 25). A C5 specification subdivides a CPCI into CPC's and describes the design of the CPCI and of each CPC (level 5b). A function in a B5 specification need not correspond to a CPC in the corresponding C5 specification although it frequently does. The combination of a B5 specification and a C5 specification is intended to describe what a CPCI does and how it does it, respectively.

Subdivision of a system into segments, of segments into functional areas, and of functional areas into CPCI's is normally, but not necessarily, based on criteria such as development by different contractors and execution on different hardware. Subdivision of CPCI's, first into functions and then into CPC's, is based essentially on software analysis and design, since an entire CPCI is normally provided by one contractor and executes on one set of hardware.

Figure 26 shows the principal events in the assumed acquisition model. The upper half of the figure relates to system events or events which are usually concerned with two or more CPCI's jointly. The lower half of the figure contains events which generally occur or are performed separately for each CPCI. The relative times shown at the bottom of the figure have been added purely for reference purposes to indicate the sequence in which events are assumed to occur. No significance is to be attached to differences in relative times or to the lengths of the periods shown in the figure. The dashed line and the lower set of relative times are intended to indicate that if shortcutting of the validation phase is attempted, then essentially all of the validation phase events (including possibly the authentication of the system specification) are assumed to occur in the full-scale development phase following selection of the development contractor, thereby lengthening the full-scale development phase.

Several relatively significant events and types of events are not shown in Figure 26, primarily to avoid more clutter in the figure. If a validation phase contractor is to be employed, he is selected early in the validation phase, in the vicinity of relative time 105. The RFP and SOW shown at relative times 150 and 210

pertain to the full-scale development contractor, who is selected following relative time 210. Not shown in the figure are the acquisition phase test and evaluation periods (development, initial operational, and operational) or the preparation of Categories I and II test plans, procedures, and reports.

The model assumes that ICD's are used to specify interfaces between CPCI's and not within an individual CPCI. Therefore, ICD's are treated in the figure as system products and not as a separate product for each CPCI.

The period between relative times 300 and 410 is a relatively busy and important one, even more so if the validation-phase events are postponed till then. The relative sequencing of the Types B5 and C5 specifications and the PDR and CDR are shown in the figure as called for by the various authorities cited earlier in this section. The authenticated B5 and parts of the C5 specifications are supposed to be available for review in advance of the PDR, and a draft C5 specification, complete except for program listings or equivalent material, is supposed to be available for review prior to CDR. Because of tight time constraints imposed in some cases by the appropriate regulations or for a particular acquisition, this ideal often seems to be difficult to achieve. The result may be some slippage of the schedules for the reviews and/or reduction of the amount of material available to support the reviews or possibly an inadequate period of time to review material before reviews.

The PQT shown at relative time 500 is only performed for critical portions of a CPCI. The system testing (Category II) shown at time 650 has been called system-level FQT by some people.

There are differences of opinion as to when initial drafts of User's Manuals (UM), Computer Programming Manuals (CPM), and Positional Handbooks (PH) are to be prepared. These are shown in Figure 26 at relative times 630 and 660, although there are at least suggestions in some of the authorities that at least one of these should be available for the CDR. A particular problem seems to be that a CPM is either intended to be used for several different purposes or different authorities make different assumptions as to its purpose. The authority which calls for it to be available prior to CDR, expects it to describe the language to be used for implementation of the CPCI. Other authorities refer to its use to describe the language implemented by the CPCI or during operational maintenance for any type of CPCI. In the latter case, it is not clear what function the CPM would serve that the combination of the B5 and C5 specifications is not intended to fulfill. In any case, the DID defining the CPC seems to need clarification or possibly separation into several DID's.

SECTION VI

METHODOLOGY DEPLOYMENT CONCEPTS

This section presents a proposed manner of deploying the SRI HDM and the Honeywell RDM in the Air Force software acquisition environment by comparing each methodology with the acquisition environment and, hence, with each other. Figure 27 summarizes the comparison. The remainder of this section consists of a few general observations about the deployment concepts and the figure. These observations are followed by a more detailed discussion which generally flows from top to bottom in the figure.

It is assumed that the principal contribution of a software design methodology is at the CPCI level, or, more specifically, that a methodology relates primarily, but not completely, to the lower half of the full-scale development phase in Figure 26. It is assumed that the basis for delineating CPCI's consists of relatively broad considerations such as development by different contractors or execution on different computers so that the software portions of the system specification are generated prior to application of either methodology. The contents of the ICD's are generally also independent of application of a methodology although it should be noted that the interface for a CPCI will generally also be described as the top-level "user" interface for use by a methodology.

The passage of time corresponds to downward progression in Figure 27. The comparison is based primarily on such things as events, products, and activities taking place in each software development phase (a block-by-block comparison rather than a line-by-line comparison in the figure). While the central column generally implies performance of each activity once for a given CPCI, the activities of the methodologies are performed for each abstract machine or module in the CPCI.

For the SRI HDM, each stage is related in the figure to the software development phases. Noted for each stage are the applicable portions of the PDL diagram (Figure 4), of the tool diagram (Figure 5), and of the example given in Figures 7 through 11. The SRI stages are grouped in three groups for comparison with three software development phases.

The two Honeywell RDM design phases (general and detailed) are compared with the same three software development phases. Noted for each RDM phase are the applicable portions of the design diagram

SRI HDM	Acquisition Environment Phases	Honeywell RDM
Stage 0 (User Interface) Stage 1 (Modularization) Figure 4 - MODULE Column, lines 1, 1.7.1 Figure 5 - enter .INTERFACE files and part of .HIERARCHY file Figure 7 - example	Validation Analysis FSD Analysis Authenticated B5 Spec. Partial C5 Spec. PDR	General Design Figure 17 - squares Figure 18 - structure, line 4 except 1.2.2.1 and 1.2.2.3, line 6 Figure 20 - both pages, thru programs Figure 23 - to line (1.2.2.2.)4
Stage 2 (Module Specs) Figure 4 - remainder of MODULE Column Figure 5 - enter .SOURCE files and execute CHECKMODULE Figure 8 - example	Design Draft C5 Spec CDR	Detailed Design Figure 17 - circles Figure 21 - outlined areas and blocks 1.2.2.1 and 1.2.2.3 Figure 20 - program stackint., program init Figure 23 - remainder
Stage 3 (Mappings) Figure 4 - MAP column Figure 5 - enter.MAP files and execute CHECKMAPSPEC and CHECKMAP, execute CHECKINTERFACE, complete .HIERARCHY file and execute CHECKHIERARCHY Figure 9 - example Stage 4 (Implementation) Figure 4 - PROGRAM_MODULE Column Figure 5 - no capability yet Figures 10&11 - example	Code	
Further non-specific contributions by above products	Checkout Test and Integration Installation Operational/Support	Further non-specific contributions by above products

Figure 27. Comparison of Methodologies and Acquisition Environment

(Figure 17), of the design documentation diagram (Figures 18 and 21), and of the two examples given in Figures 20 and 23.

In Figure 27, the dashed line near the top of the acquisition environment indicates possible contributions by both methodologies to the analysis phases of both the validation and full-scale development phases. The dashed line near the bottom of the acquisition environment indicates contributions of the methodologies primarily to the coding portion of the code and checkout phase. For checkout and the last 3 phases of the acquisition environment, the methodologies provide no new activities or products; the products produced in the earlier phases are used to supply information to support these later phases in a general way.

ANALYSIS PHASE

The analysis phase of software development is assumed to be split into two components - one performed during the validation phase of system acquisition and the other during the full-scale development (FSD) phase. It is quite possible for these two analytical efforts to be performed by different organizations - the validation effort by a contractor or by PO personnel and the FSD effort by a second contractor. A principal product of the validation phase is a preliminary version of the B5 specification for a CPCI, produced by one organization. The B5 specification is authenticated in the FSD phase, which implies, among other things, that the second organization approves it for use as a guidance document.

A B5 specification is claimed to be a requirements document and not a design document; it should specify <u>what</u> a CPCI is to do and not <u>how</u> it is to do it. However, a B5 specification segments a CPCI into "functions", each of which is described in terms of its inputs, processing, and outputs. Therefore, a function seems to have the characteristics of a structural element of a CPCI rather than being just a convenient way to group a set of related requirements. While a CPC of a C5 specification need not coincide with a function of the corresponding B5 specification, it seems reasonable to assume that the delineation of functions will frequently have a significant influence on the later delineation of CPC's. Therefore, it is assumed that, regardless of what else it is, a B5 specification represents a preliminary, general design for a CPCI, which is further detailed, possibly with changes, in a C5 specification. It is also assumed that if a B5 specification is a design document and if a design methodology is to be used to design the CPCI, then the

methodology should provide the basis for, and as much of the content as possible of, a B5 specification.

A dilemma must then be resolved. Essentially everyone is agreed that it is undesirable, if not illegal, to require a contractor to use a particular methodology (see Section VIII, in particular). If a methodology is chosen by the first organization for the validation effort, then this methodology should affect the B5 specification which is to be authenticated by the second organization during full-scale development. Cost-effectiveness considerations suggest that this methodology be used for the remainder of software development. Resolving this dilemma is not within the scope of this effort. The larger issue, which should be addressed in the future, is to determine how benefits of design methodologies can be achieved and still provide for easy transition from one contractor to another, preservation of competition, and cost effective development.

With the present regulations, it is assumed that a function in a B5 specification will correspond to a module and that the early stages/phases of a methodology will provide the basis for the design aspects of the B5 specification. For the (SRI) HDM, a B5 specification should contain a listing of the modules comprising each abstract machine and the hierarchy of abstract machines. Such a listing is similar to a set of INTERFACEs and one HIERARCHY as described under Tools in Section III except that the counterpart of an INTERFACE should define only the design aspects of an abstract machine (and no implementation aspects) and the counterpart of the HIERARCHY would not include the mappings. For the (Honeywell) RDM, one might want to list the inter-module relationships although these relationships are contained in the formal products from the general design phase (the abstract data types in the "types" block of Figure 18).

The remainder of the information that the methodologies can provide for a B5 specification is organized (by the methodologies) on a module basis. In a B5 specification, all of Section 3.2, Detailed Functional Requirements, except for 3.2.n, Special Requirements, is organized on a function basis. Each function is described generally and then it is defined in terms of its inputs, processing, and outputs. Since a function is assumed to be a module, the characteristics of a module which have been identified during the "general design phase" should be used to describe/define a function.

For the HDM, the "general design phase" consists of stages 0 and 1 of "design and implementation", yielding products such as

shown in Figure 7. In addition to the INTERFACEs and HIERARCHY discussed above, these products define the formats and names of V-, O-, and OV-functions and their input arguments and result arguments. It is possible to perform a trivial mapping between Figure 7 and a subsection of Section 3.2 of a B5 specification by logically ORing all input arguments and treating them as inputs, by treating the collection of O- and OV-functions as processing requirements, and by treating the collection of result arguments as outputs. Such an exercise seems undesirable from several viewpoints, however. The representation of a module in terms of its state variables and operations and the input arguments of each seems to have a greater information content than if this representation is artificially dismembered into inputs, processing, and outputs. Furthermore, it appears that stages 0 and 1 of the HDM are not intended to yield much more information than that contained in Figure 7 plus information which can be gleaned from the choice of names for functions and parameters plus a relatively short comment associated with each HDM function plus diagrams such as Figure 6. Further fleshing out of this information involves stage 2 and detailed design. Therefore, under the HDM, Section 3.2 of a B5 specification should probably be organized differently and contain more highly distilled information content than at present.

The general design phase of the (Honeywell) RDM has some advantages and some disadvantages over the HDM, at the module level. The RDM seems to develop somewhat less information at the operation or program level. Each program is named and can be described by comment in the "programs" block on line 6 of Figure 18. Parameters of the programs at present are not defined directly, only in comments or in an abstract data type declaration of a higher-level module which uses the lower-level one. General design also defines data which are potentially global to the module as well as relationships with lower-level modules by means of abstract data type declarations. Data invariants, if any, are defined. Dismembering a module into inputs, processing, and outputs is probably again inappropriate. The top-level data of the module are relatively well developed, but lower-level data and all operations are defined further in the detailed design phase. The "functional-description" block for each module on line 4 of Figure 18 should provide a general description of each module for inclusion in Section 3.2 of a B5 specification.

The remainder of a B5 specification (except for the first subsections of 3.2) is at the CPCI level rather than at the function level. Neither methodology provides any significant collected information at the software system level (corresponding to the CPCI level). The user interface for the top-level module or abstract

machine corresponds to the CPCI external interface (Section 3.1.1 of a B5 specification) but this interface is an input to a methodology rather than a product of it.

The general structure for design documentation and various text blocks for the RDM shown in Figure 18 can provide information for a B5 specification. Note that except for line 1 and the "module-index" on line 2, all blocks pertain to a single module. To obtain information at the CPCI level, the blocks for the top-level module would have to be used to describe the entire software system rather than the top-level module, or information would have to be collected from corresponding blocks of all modules and blended together. Each alternative has drawbacks. Since the basic framework has been defined, a preferable alternative may be to define additional blocks at the software system level. The "acceptance-criteria" block for each module on line 4 of Figure 18 should supply information for Section 4, Quality Assurance Provisions, of a B5 specification. Other blocks should contribute to a C5 specification and a User's Manual, discussed below.

One aspect of the RDM could require modification of the acquisition environment. If, as suggested by Honeywell, it is necessary to do detailed design of some of the higher-level modules before completing general design of all the lower-level modules, then detailed design would have to start prior to assembling all necessary information for the B5 specification for an entire CPCI and prior to holding a PDR for the entire CPCI. Either the B5 specification and the PDR would have to be done in several increments or they would both have to follow the beginning of detailed design.

DESIGN PHASE

One of the most significant products which is intended to result from the design phase of the acquisition environment is the C5 specification. In a C5 specification, Section 3.2, Functional Description, is used to detail each CPC and the remaining sections pertain primarily to the CPCI as a whole. The methodologies make varying contributions to a C5 specification. To the extent that a module corresponds to a function in a B5 specification and to a CPC in a C5 specification, the effort to produce Section 3.1, Functional Allocation Description, of a C5 specification is minimized.

The detailed design phase for the HDM is considered in this report to consist of stage 2 which produces a specification for each module such as those in Figure 8. SRI seems somewhat ambivalent as

AD-A056 771 MITRE CORP BEDFORD MASS F/G 9/2
SOFTWARE DESIGN METHODOLOGIES AND AIR FORCE SOFTWARE ACQUISITIO--ETC(U)
JUN 78 D L JAMES F19628-77-C-0001
UNCLASSIFIED MTR-3508 ESD-TR-78-147 NL

2 OF 2
ADA
056771

END
DATE
FILMED
9 78
DDC

to whether they consider stage 3, which produces mappings, to be a design or an implementation activity. It seems preferable to consider mappings part of implementation since they really relate to the representation of abstract machine Mi in terms of machine Mi-1.

The intent and the world view of a C5 specification seem to differ from those of the methodologies in at least several respects. One of the more significant differences relates to Section 3.3.1, Data Base Characteristics, of a C5 specification. The intent of this section is to describe in detail all of the data base of the CPCI which is not specific to a single CPC and to include such things as a set-used matrix to indicate which CPC's access which data elements. Some of this information would not be supplied until after implementation in the acquisition environment. On the other hand, AFR14-2 calls for availability of the "structure and organization of the data base" portion of Section 3.3.1 of a C5 specification prior to PDR.

The general view of the HDM and the RDM (as well as other methodologies) is essentially that there is no CPCI (software system) data base. The data base is spread over the modules. A module is responsible for a specific portion of it and is the only module which may access that portion directly. Furthermore, other modules and their designers and implementors should know only the minimum, necessary details about that portion of the data base. Other modules are to read that portion of the data base and modify it only by a means such as making procedure calls to the responsible module which will fetch and deliver a data value or execute an operation. While it would be possible, after implementation, to extract all the data base details from all the modules and consolidate them in a Section 3.3.1, it seems undesirable to do so. With easy, central access to all the details of the data base, a designer or programmer doing program maintenance during the operational phase could introduce intermodule dependencies, which were carefully avoided during the design phase, by making the design of one module dependent on the current implementation of another module, for instance. If a particular mode of operation and level of information are deemed desirable during the design phase, then it is probably desirable to preserve them during the operational phase as well.

The desired, more abstract view of the data base could be extracted from the modules and put in Section 3.3.1, but it would probably merely duplicate much of the contents of Section 3.2.

It is quite possible that there are some practical, data-related problems not addressed by the methodologies. Regardless of

how neatly compartmented a design is, for a large system it may be difficult to remember or locate responsible modules and names and formats for procedure calls so that some form of cross-referencing between modules and data may be desirable. Cross references to indicate who calls each procedure are probably also desirable. The RDM feature of allowing data to be declared of type abstract (meaning that its specific type is declared elsewhere) might create problems of implementation as well as later understanding.

Section 3.2.1.3, Interfaces, of a C5 specification, also calls for a set-used matrix to relate each CPC to the CPCI data base and to other CPC's. Such information appears to duplicate at least partially that for Section 3.3.1; the comments above apply to Section 3.2.1.3 as well.

Flow charts are called for at the CPCI level in Section 3.4, Computer Program Functional Flow Diagram, and at the CPC level in Section 3.2.1.2, Flow Chart, of the C5 specification. Flow charts are in relative disfavor because of the amount of effort to produce them and the likelihood of their becoming out of date. With the expected small size of modules, flow charts are less useful. With software designed using the HDM or the RDM, a flow chart is apt to reflect primarily a portion of the hierarchy chart relating the modules. Such a chart was discussed as part of the B5 specification.

An important part of detailed design for both methodologies is the specification of the formal requirements for each operation (program). These requirements should probably be included in the C5 specification since they are developed after the B5 is done. The EFFECTS section of an O- or OV-function of the HDM specifies what action the function is to perform in terms of the changes it produces on the state variables. For the RDM, the "input-requirements" and "output-requirements" blocks for a program specify the set of expected states of the module at entry to the program and the set of required states at exit, respectively.

For the RDM, the "design-overview" block (for a module) on line 4 of Figure 21 should provide information for Section 3.2.1.1, Description (of a CPC). Section 3.2.1.4, Data Organization, of a C5 specification is to describe the internal data of a CPC. The detailed design phase of the RDM provides for detailed definition of the program of a module. In the process, program refinement is used to define internal program blocks and possibly further internal data variables. It is probably better to document such variables with the program blocks rather than collect them into one section such as

3.2.1.4. (Section 3.2.1 and the six subsections it contains are to be "repeated" for each CPC.)

The HDM PDL, SPECIAL, is a non-procedural PDL. However, the RDM PDL is procedural in nature. As Figure 27 indicates, the RDM detailed design phase includes most of the effort of implementation as well as design. Therefore, it would not be possible to complete the C5 specification and hold a CDR before implementation starts. In conjunction with the comments under the Analysis Phase, the RDM could result in implementation of the higher-level modules of a CPCI being essentially complete before a PDR is held for the entire CPCI.

CODING PHASE

Stages 3 and 4 of the HDM are assumed to correspond to the coding phase of software development. The products of these stages would furnish the implementation portions of the C5 specification. Other relevant comments were made previously.

OTHER PHASES

In general, the methodologies are not intended to provide any specific, new products to benefit these later phases. The products from the earlier phases and the organization of the design are expected to contribute generally to reduced effort in program debug and test and system test as well as later operational maintenance activities. For the RDM, the collection of "usage-information" blocks for all modules in line 4 of Figure 18 should contribute to production of a User's Manual for the CPCI in the test and integration phase.

SECTION VII

INFORMATION GATHERING METHODS

The activities of this subtask that were of prime importance to RADC were to hold discussions with experienced software acquisition personnel (from ESD and MITRE) of the SRI HDM, the Honeywell RDM, and the possible application of such methodologies to the Air Force software acquisition environment, to obtain feedback from the personnel about the application of methodologies, and to record the feedback in this report. The emphasis was to be on the effects on Program Office activities of the use of methodologies (rather than on contractor activities) and on necessary or desirable changes to the methodologies and/or to the acquisition environment to provide a better match between the two. The other activities of this subtask were performed to support those listed above.

It had originally been hoped that the discussions could elicit feedback on the opinions of acquisition personnel as to the advantages and disadvantages of individual features of one methodology versus the other. It was felt that it would be presumptuous to attempt to involve any individual from the acquisition environment in such discussions for more than approximately half a day and that a relatively large portion of that period would probably have to be devoted to discussion of the methodologies per se. Background material was distributed to the two groups of discussion participants approximately a week or more before the discussions were held so that as much time as possible might be devoted to discussions of the application of methodologies in the Air Force environment. This background material consisted essentially of Sections II, III, and IV of this report and reference Lind76. (The purpose of the discussions had been outlined previously by memo.) However, it was recognized that the complexity of the methodologies, the lack of very extensive advanced publicizing of the methodologies by their proponents (particularly true for the Honeywell RDM), and the assumed length of a discussion would make it extremely difficult to explore many individual features of one methodology versus the other in the discussions. In fact the outline for the discussions of the methodologies per se had to be shortened several times to allow reasonable time for discussion of their application in the Air Force environment.

Two discussions were held, each with a separate group, near the end of August 1977. The two groups consisted of ESD and MITRE personnel from the TACC Automation project and from the SATIN IV

project. The discussions were each approximately three hours long, with approximately half of each devoted to a presentation of methodologies and the other half to group discussions of their application. The TACC Automation discussion was held first. With limited discussion it was possible to present the SRI HDM and an example of it before the group discussion began. Consequently, for the SATIN IV discussion, the presentation was shortened even further so that the characteristics of the methodologies were presented primarily in terms of the examples, with greater attention paid to the Honeywell RDM than to the SRI HDM.

The feedback from discussion participants on the methodologies and their application in the Air Force software acquisition environment is given in Section VIII. Shorter discussions of the acquisition environment only were held with several MITRE people near the end of May 1977, and several comments from these discussions are also included in Section VIII. Section IX contains primarily comments of the author of this report on various individual features of the two methodologies.

PRESENTATION TO ACQUISITION PERSONNEL

The following material summarizes the full presentation prepared for the discussions with acquisition personnel. As noted above, different versions were used for the two groups.

Figure 28:

Summarize project subtask.
Summarize purpose of meeting.

Figure 29: Summarize characteristics of software design methodologies.
Finite-state machine (FSM):

- Current state = current values of <u>state variables</u>
- <u>V-function</u> returns current value of a state variable
- <u>Operation</u> changes certain state variable values: <u>O-function</u>
- Effect of O-function defined only in terms of state <u>changes</u> it produces
- V-function returns no value, O-function causes no state

SUMMARY

PURPOSE OF PROJECT SUBTASK

PURPOSE OF THIS MEETING

CHARACTERISTICS OF SOFTWARE DESIGN METHODOLOGIES

ABSTRACT DATA TYPES

SRI EXAMPLE

HONEYWELL EXAMPLES

ASSUMED ACQUISITION ENVIRONMENT

USE OF METHODOLOGIES IN ACQUISITION ENVIRONMENT AND DISCUSSION

Figure 28. Presentation Summary

CHARACTERISTICS OF METHODOLOGIES

DOCUMENTATION

APPLICATION

INCREASED DESIGN EFFORT

DEVELOPMENT PHASES

COMPUTER STORAGE OF DESIGN

FORMAL DESIGN SPECIFICATIONS

USE OF PROGRAMMING DESIGN LANGUAGE (PDL)

TOOLS

GUIDELINES

Figure 29. Summary of Methodology Characteristics

change if exception condition is true

- V-function or O-function can have parameters
- An FSM should have exclusive control and knowledge of a kind of data and the operations available on it
- Lower-level FSM's implement a higher one (or help to implement it) (serve as implementation base)
- The visible interface (variables and operations) of an FSM defines an abstract data type
- An abstract data type defines and implements objects (which occur at a higher-level and are used at that level) as well as operations upon them
- An FSM is composed of one or more modules

SRI methodology largely formal, to support proofs. Each abstract machine provides base for implementation of next higher one.

Figure 1

Figure 2

Define V-, O-, and OV-functions.

Figure 30

Figure 6: Define the segment.

Figure 7: Stage 0 - user interface.

Figure 6 (again): Define page array, page pool.

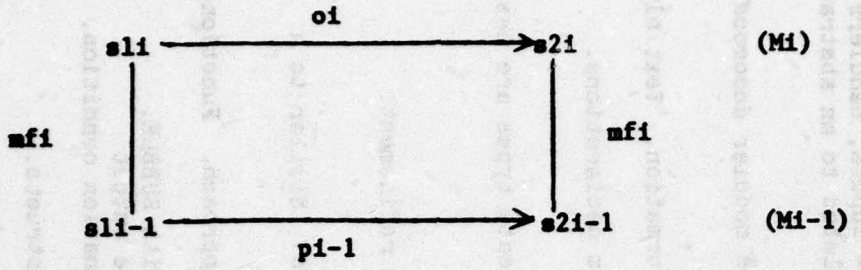
Figure 7 (again): Stage 1 - modularization.

Figure 8: Stage 2 - module specifications for each module.

Figure 9: Stage 3 - data (variables, PARAMETERS) mappings between modules.

Figure 11: Stage 4 - implementation of bottom-level O-functions on computer. V-function names reference storage locations.

SRI METHODOLOGY



STAGE 0 - USER INTERFACE

1 - MODULARIZATION

2 - MODULE SPECIFICATIONS

3 - MAPPING OF MACHINE Mi DATA IN TERMS OF MACHINE Mi-1 DATA

4 - IMPLEMENTATION OF MACHINE Mi ON MACHINE Mi-1

Figure 30. Overview of SRI Methodology

Figure 10: Stage 4 - implement higher-level abstract machine on lower-level one.

Figure 4: Brief discussion of SPECIAL.

Figure 5: Brief discussion of SRI tools - inputs, handlers, errors, consistency. INTERFACE is related to an abstract machine.

Figure 17: Honeywell methodology provides modular decomposition and program refinement.

Figure 16: Textual as well as formal information. Text blocks and framework for other blocks.
General design - data mostly. Program declarations.
Data declarations.

Figure 20: Type declarations. Abstract data types are basis for modular decomposition.
Detailed design.

Figure 23: BUBBLE sort example. Program refinement.
Program variables.

Figure 19: Array variables and operations. Similar to a built-in abstract data type.
Program requirements.
Program design logic. Constructive approach. Functional forms.

Figure 23 (again): Constructs - define with BUBBLE.
Guarded statements. For do, decompose output requirements into invariant and termination condition.

Transformation characteristics of constructs.

Tools: None yet for RDM, hope to adapt those of WELLMADE which provide limited support, investigating tools for syntax checks.

Figure 25: Assumed acquisition environment.

Figure 26: Major events. Software events (one CPCI) separated from system events (two or more CPCI's jointly). Dashed line - apparent effect of limited validation phase.
Methodologies primarily affect lower half of FSD phase.

Discuss operational concept for use of methodologies in

acquisition environment (compare methodologies with environment, as in Figure 27). Elicit feedback from participants with questions.

DISCUSSION SUBJECTS

Questions were formulated to use in the discussions following the presentations to acquisition personnel in an attempt to elicit their opinions of the methodologies and application of them in the acquisition environment. The subject areas included:

1) PO organization changes needed for use of methodologies,

2) Education and training needed by PO personnel,

3) Experience needed by PO personnel,

4) Desirability and effect of increased design efforts called for by methodologies,

5) Extent of design monitoring desirable and feasible,

6) Extent to which methodologies support design monitoring by an organization separate from the designers,

7) Views on computer storage of a design and continuous monitoring of such a design,

8) Understandability of requirements expressed in terms of state variables and of products of methodologies, particularly to those with approval responsibility,

9) Ease of transfer of a partial design developed using a methodology from one organization to another,

10) Desirable form and extent of definition of a methodology,

11) Desirable changes to a methodology or the acquisition environment, and

12) Acceptable trade-offs between system software testing and performance of proofs-of-correctness of software.

In general, it was not possible to discuss all of these subjects and others not listed with acquisition personnel, primarily due to limitations in their time which precluded more detailed

presentation of background material and more extensive discussion of use of methodologies in the acquisition environment. Sections VI and IX attempt to cover the opinions of Project 522M personnel in such areas. In particular, several subject areas which had been suggested previously were not explored with acquisition personnel. Proof-of-correctness was not explored since it would have required much more time than available to present and discuss due to its complexity. Utility of and possible changes to tools to support methodologies were not explored largely for the same reason, although tools are less complex than proofs. Adequacy of present documentation of the methodologies and kinds of improvements desirable were not covered since these subjects seemed less important than others and ones which Project 522M personnel could evaluate.

SECTION VIII

ACQUISITION PERSONNEL FEEDBACK

The following represent the principal opinions expressed by ESD Program Office and MITRE personnel, pertinent to the use of software design methodologies in the Air Force software acquisition environment. Although these opinions have been discussed and expressed here by RADC and Project 522M personnel, we have attempted to avoid coloring the opinions of others by our opinions or by our manner of expressing them. Our comments, if any, in parentheses, follow each opinion.

The TACC AUTO and SATIN IV feedback are based on relatively lengthy group discussions of the SRI and Honeywell methodologies and the acquisition environment held on August 29, 1977 and September 1, 1977, respectively. The PAVE PAWS and JTIDS feedback are based on shorter, individual discussions of only the acquisition environment held on May 26, 1977, and June 1, 1977.

TACC AUTO

- If a particular methodology is intended to be adopted as an Air Force standard, contractor resistance as well as Air Force resistance should be expected on the grounds of stifling of competition.

  (It was pointed out that the selection of a particular methodology as an Air Force standard is not the intent.)

- It is not the Air Force's business to require that a contractor use a software design methodology. If a particular bidder believes that a particular methodology will help him, he should and will propose its use in his proposal.

  (The emphasis here is on requiring the use of some methodology, not a particular one. Requiring that a contractor use some systematic method of designing software may not be inconsistent with the other requirements levied upon him by the acquisition environment. Source selection policies which favor low bidders for software development may discourage or effectively eliminate proposals which increase the developer's design costs even though the increase may be offset by later savings in software maintenance by the user. Therefore, if one wants to

ensure the use of a methodology, it may be necessary to require it and expect that design costs will increase.)

- Due to time, manpower, and budget limitations, the PO should concentrate on understanding the requirements and the process to be used to produce a design, rather than on the design itself, i.e., audit the design process rather than the design, to save manpower.

  (This view may be predicated on the assumption that another organization such as MITRE is involved in an acquisition and is responsible for understanding and evaluating the design.)

- The PO needs data indicating the effect of the use of a methodology on project costs and schedule.

- It must be proved at a technical level that a methodology works on real problems. A demonstration, a New York Times-type project is needed.

  (The emphasis on a New York Times-type project may imply more than simply a contracted demonstration. A spontaneous model project proposed or adopted by a contractor, is probably far more likely to convince a PO to use a methodology than is a directed demonstration in which a contractor is simply paid to use particular methods.)

- The usual type of PO member with a BS in engineering may have an inadequate background in software design for large systems and, therefore, may not be able to understand the use of a methodology such as SRI or Honeywell and the formal designs they produce. A degree in computer science may be necessary.

  (If an engineer does understand software design, programming, and the use of higher-order programming languages, however, he may only need further training and experience in the use of design methodologies rather than further formal education. With proper documentation and good examples to study as well, such people may achieve adequate understanding of proof methods as well.)

SATIN IV

- being too specific in requiring a successful bidder to use a software design methodology may stifle competition. In addition, such specificity may result in the choice of a

methodology which is inappropriate for the system to be acquired.

- If a contractor is convinced that the use of a particular methodology will benefit him, you won't be able to prevent him from using it. On the other hand, if a contractor believes the use of a methodology will increase his risk, it will be difficult to convince him to use it. A contractor may expect to be paid to train his people to use a new methodology. If a project is not completed or runs out of money, a methodology associated with it may acquire a negative image.

- With regard to willingness to accept the increased design efforts suggested by many methodologies, some of those present expressed willingness if a better system would result. Others emphasized willingness to accept higher costs in return for increased scheduling accuracy, reduction of risk, earlier identification of potential schedule slippage, etc.

- Initial use of a methodology may be hard to sell. Early use of a methodology on small projects may be helpful; however, in applying past experience to a new project, similarities in type and size of system are important. People may be willing to accept techniques which were successful on a previous project for use on a new project which is three times the size of the previous one but require more evidence of applicability if the new project is ten times the size of the previous one.

- Structured programming (SP) was chosen for use on their project because of (1) belief that SP had relatively widespread acceptance, (2) belief that use of SP would contribute to proving the correctness of code for multi-level security control, and (3) statements in an RADC paper that programmers voluntarily adopted the use of SP as a result of association with other programmers who were required to use SP.

- There is disagreement between individuals and between PO's as to the extent of design monitoring desirable. The number of CPCI's involved may preclude detailed monitoring. Monitoring of the process by which a design is produced and of the top-level design may be preferable to monitoring the complete design. One purpose of holding reviews is to determine the contractor's understanding of the requirements and his apparent assurance as to the adequacy of his design. Design monitoring may be compared with the activities of a financial auditor who is not attempting to find small discrepancies but procedures which allow systematic discrepancies to go undetected.

- With regard to the use of a terminal by a PO to monitor a design stored in a computer, their contractor proposed their use of one. If used, it should be used only to obtain top-level management and status reports, not for following day-to-day progress or for browsing through the design data base. The contractor is getting paid to do the management and monitoring of the detailed work. The PO understanding of the design and of work progress should be based on formally submitted reports and not on information obtained by other means. It is usually undesirable for a PO to know more about the design than the contractor management. The PO may make incorrect assumptions based on incomplete or temporary data in a design data base.

- Tight timing constraints, particularly shortly after contract award, can result in products such as the authenticated System Specification and the Computer Program Development Plan failing to satisfy their intended purpose and/or attempting to satisfy the purpose of other documents.

  (The application of increased effort in the early stages of use of a methodology could further exacerbate such problems unless timing constraints are relaxed.)

- The highly condensed form of the early products from some design methodologies does not necessarily represent a change from present practice. Contractors at times supply little if anything more than the names of software components.

- It can be difficult in current-day Air Force specifications to trace requirements (locate the response in a specification to a known requirement) and to trace data flow (determine from a specification all the processing a particular input is subject to).

  (TRW's SREM - Software Requirements Engineering Methodology - specifically aims to provide solutions to these two problems. The SRI and Honeywell methodologies provide less specific help in these areas although a design which is modularized by clustering the processing around each kind of data object and by limiting access to each kind of data object should make it easier to trace requirements and locate the processing for a particular input.)

- SATIN IV changed the normal documentation requirements with DID backup sheets to do such things as incorporate HIPO diagrams and PDL representations in specifications and remove interface specifications to separate documents.

(This route could be used to tailor/change contractor-product requirements in consonance with a particular class of methodologies or a specific methodology, for a demonstration project or pilot project, for instance.)

- Timing constraints and insufficient knowledge on the part of the person responsible for software documentation can result in inadequate documentation at present.

  (The software design methodologies are intended to produce software documentation as an integral part of the design process rather than as a separate effort after the fact, so that documentation should be prepared by knowledgeable people, at the time when it is freshest in their minds and should be up-to-date. If a methodology produces a better-organized design, the documentation of the design should be more comprehensible also.)

- Contractors sometimes have trouble in determining which design decisions need to be made and which of these are most critical. The recording and communicating of critical decisions can also be a problem, particularly when parallel efforts are in progress and for design areas which affect or overlay several others, such as system control, error recording and reporting, and interrupt handling. Changes in management practices may reduce but probably not eliminate such problems.

  (Some methodologies claim to facilitate or foster the recording of critical design decisions. Elaboration of such claims and the provision of guidelines seem to be lacking at present. However, any aspect of a design recorded in a computerized data base is potentially communicatable to other interested parties; automated tools to trigger appropriate reports when changes to such a data base are sensed could be useful, if available.)

- If the use of a methodology places greater reliance on formal specifications which may require specialized knowledge to understand, it may be desirable to reduce the distribution of copies for review. A reduced distribution would not eliminate useful feedback in the case of those external agencies which currently fail to respond to requests for review of specifications.

- A methodology which provides guidelines to contractor and PO personnel as to the content and structure of the top-level design for presentation at a PDR, for instance, would be beneficial.

(A number of methodologies address this problem, although further design guidelines and increased content of specifications seem desirable in some cases.)

- The SRI and Honeywell methodologies don't start early enough.

  (They don't address requirements analysis and requirements traceability. Even SREM seems more oriented toward recording results in these areas than in providing guidelines.)

PAVE PACE

- The present Air Force software acquisition process seems reasonable and can produce good systems. Where problems arise, they can frequently be traced to attempts to bypass steps in the acquisition process; in particular, shortcutting the validation phase can cause troubles.

JTIDS

- The present Air Force software acquisition environment might be improved by providing for more continuing review of contractor efforts by the PO, in particular between CDR and FCA, for instance.

  (Presumably this problem will seem worse for long projects than for short ones. The problem is worsened if there is a tendency to produce skimpy B5 specifications and not produce draft C5 specifications until coding or even testing is relatively complete. Various authorities differ as to how early in the software development cycle the C5 specifications should be available, some calling for early sections in time for PDR, and the completeness of draft C5 specifications for CDR. If problems in meeting schedules occur, there seems to be a tendency for reviews and draft specifications to be more perfunctory than intended. If a staged methodology, such as the SRI one, is used, more frequent reviews could be required based on the product from each stage.)

SECTION IX

PROJECT PERSONNEL OPINIONS

This section presents opinions of project personnel, primarily the author of this report, on various aspects and features of the HDM and the RDM. Since Section VI and the parenthesized comments of Section VIII tend to reflect our views on the suitability of the two methodologies for use in the Air Force software acquisition environment, this section concentrates more on their general characteristics, with possibly greater attention to the HDM, since documentation of it is more extensive.

GENERAL

Methodologies need guidelines for their use. Presumably further guidelines will evolve with greater use of the methodologies. Guidelines for starting a design seem relatively weak. The form, content, and organization of the input to the design process are generally ignored. Examples of use of a methodology tend to emphasize the resulting design with minimal attention to the requirements which dictated that design. Both the HDM and the RDM produce formal requirements in the detailed design phase in terms of state variable values. Such requirements are highly distilled and structured and presumably are not just extracted from a requirements document such as a typical Type A specification. Because of the design orientation of the formal requirements, it can't be argued that their preparation is a part of requirements analysis, and, therefore, not properly a part of a design methodology.

Guidelines which suggest a staged/phased development need to avoid suggesting that the stages/phases are relatively independent of one another. Both the HDM and the RDM produce in general design a modularization and a hierarchy, a set of potential dependencies between modules and functions of modules. Only in detailed design and implementation are the actual dependencies specifically stated. To minimize the looping back from detailed design and implementation to make general design changes, one probably has to consider some aspects of detailed design and implementation while doing general design. The RDM suggests that some of the detailed design and implementation may need to be performed before general design can be completed. It may be that for a complex software system additional tools may be needed to record the existence of relationships and

dependencies between software elements, to look for conflicts and to form hierarchies. Existing and planned tools will record relationships and detect conflicts as by-products of detailed work in later stages but earlier assistance may also be necessary.

The kinds of information which the HDM and the RDM record, the form of those recordings, and the kinds of information not recorded seem to be tailored to use by a relatively fixed group of people who are more or less in continuous contact with the development process. Such information may be inadequate for software monitors who have only periodic contact with parts of the process and the design or to a new group of people which takes the place of the old group as a result of a new contract for the next acquisition phase, for instance. Although the RDM provides for specific types of text blocks and the HDM only for comments, both methodologies provide BNF representations of their design documentation. The BNF could be extended to allow specific text blocks to be included in the design documentation which are more specifically oriented toward the Air Force acquisition environment. Tools could be changed or added which would signal the absence of such text blocks.

It may be desirable to alter the content and makeup of the B5 and C5 specifications, which seem somewhat more oriented to hardware than software. A product specification for hardware presumably is more often used to allow a new contractor to be selected who will produce hundreds or thousands of copies of a product or to assure that in the case of multiple-sourcing, each contractor's product is interchangeable with another's. A "product" specification for software is at least partially oriented toward after-the-fact documentation of the content of a single "prototype" (which can be mass produced by a trivial copying process) and toward documentation to facilitate future changes. The B5 specification might be split into a requirements document with little design-oriented structure and a general design document which structures the requirements and represents initial design. The first piece would be input to a methodology and the second piece would be produced with the aid of the methodology.

The C5 specification might be split into one which documents the detailed design and another which documents implementation. The possibility of doing some detailed design and implementation before completion of general design also needs to be taken into account as does the possibility of concurrent detailed design and implementation with some methodologies such as the RDM. If the software specifications are split into more pieces, the orientation should clearly be toward forming a complete document as the sum of

the pieces rather than increasing the effort required by attempting to make each piece self-explanatory.

The HDM and the RDM both suggest that some attention has been given to design of systems involving such concepts as shared data, concurrency, parallelism, multi-processing, etc. The ability to deal with such systems is certainly needed. Such capabilities appear to require further development for both methodologies or at least need to be better publicized if already fully developed. Dijkstra's guarded commands which form a part of the RDM seem to be oriented toward possible execution in parallel, although the actual intent does not seem to be enunciated.

Most methodologies, including the HDM and the RDM, are primarily oriented toward producing designs which satisfy functional requirements rather than performance requirements. A few methodologies suggest that they can be used to produce good designs that are independent of the exact placement of the interface between hardware and software, that it should be possible to convert hardware functions to software functions or vice versa, particularly in the vicinity of the interface, without changing the design. Whether such statements can presently be substantiated or not, the availability of inexpensive yet powerful microprocessors and the tendency to interface multiple processors of various sizes in the same system seem to suggest an increased number of design alternatives. When each CPU costs a million dollars, a few hardware configurations may be considered and then a large amount of effort may be spent in designing the software to run efficiently on a single CPU or each of several. If a variety of CPU's is available for widely varying prices and if the use of more CPU's with a lower average price is considered advantageous, more alternatives must be considered. Methods suited for determining whether one or possibly two or three larger CPU's are needed are probably unsuitable when a microprocessor can replace fixed logic and smaller CPU's can be used in series and/or parallel to replace larger ones. Hardware/software trade-offs become more important and more numerous. Selection or design of hardware and software architectures become more closely related. The hardware needed for a system can be more closely fitted to the requirements, with reduced expense for idle capability. If a "software design methodology" can only be used after requirements have been allocated to hardware, then the utility and need for such a methodology will be reduced as the set of requirements is apportioned over a larger number of smaller CPU's. The method of apportioning, timing, interfaces, paths between CPU's become more important and the efficiency of utilization of an individual CPU less important. Design methodologies may need to emphasize solutions to problems such as these in the future.

The effect on the acquisition environment of more CPU's with smaller programs in each must be considered. Holding separate reviews and writing separate specifications for the software in each CPU could increase costs with little increased benefit.

For the HDM, the characteristics of an OV-function need clearer delineation. Current documentation of the HDM tends to make statements about V-functions or O-functions while failing to clarify whether the statements also apply to OV-functions, and if not, what statements can be made about OV-functions. In particular, it is not clear if the result returned by an OV-function can, cannot, or may be a state variable.

Robi77 states that the differences between specification and implementation are often so great that the specifications cannot be construed in any way as a guide to efficient implementation and separate information must be supplied. Proof of the implementation would seem to be impeded or prevented in such cases. The elegance of the methodology is reduced, and traceability from code to requirements hindered. If hardware efficiency considerations become less important for less expensive CPU's, then less need would be felt for departing from the design in implementation.

LANGUAGES

The following comments relate to the PDL's (of the HDM and the RDM), definition of their syntax by BNF, and their use. Note that the BNF definitions in both cases include the design structure and design documentation as well as the PDL so that the exact boundaries of the PDL may be debatable. Some of the comments also apply to the SRI implementation language, ILPL, which incorporates elements of SPECIAL by reference.

One problem with these languages is that they are defined primarily by use of BNF - concentration on syntax to the detriment of semantics. The semantics of much of the SRI PDL (SPECIAL) have been formally documented in Roub77 and Robi77, but SPECIAL is a rather extensive language and the meanings of some of its more esoteric features are still unclear. The interpretations of some of the uses of several special characters are not clear. As a result, at least a few very lengthy statements (15 to 20 lines) in Neum77 are virtually incomprehensible. ILPL is defined only in BNF. The semantics of the RDM PDL are largely undefined as yet.

The BNF for each of the three languages tends to treat a long list of constructs as alternatives of the syntactic element

<expression> and then relies implicitly on type checking to prevent invalid combinations of them with one another and with various operators. Any feasible subsetting of expressions and operators by type would probably make the BNF definitions more meaningful even though more syntactic elements would result.

For certain applications (such as this project, infrequent references, language comparisons), the BNF definitions by themselves appear to be time consuming to use. Numbering the definitions and providing a directory of syntactic element definitions, of reserved words, and possibly of references to syntactic elements could be helpful. Defining a syntactic element whose name is the plural of another emphasizes the relationship between the two but also requires any reference to either to be examined closely enough to avoid mistaking one for the other. Failure to include BNF definitions for the more basic elements of a language can cause fruitless searches until their absence is recognized.

The Honeywell RDM PDL is relatively difficult to understand in the area of type declarations, particularly in the area of abstract data types, illustrated by the declaration for stack at the top of Figure 20. To identify the kinds of syntactic elements involved, one can look for the reserved word types in the BNF syntax definitions. It seems to appear only once. The element following the word types leads one (through an intermediate definition) to the definition for a type declaration which starts with "type <id>:". To verify that the 10 lines in parentheses which follow the ":" are an object type, one generally has to follow through five more definitions. Some of these definitions consist of 6 and 8 alternatives each. The alternatives are reserved words, synonyms for reserved words, and other syntactic elements. Separation of the 10 lines by semicolons indicates that the 10 components are a product list - that stack consists of all 10 components - rather than a union list. The outer parentheses are optional. The first line is another type declaration (within a type declaration). If the reserved word type in front of "element" is omitted, the meaning is unchanged but the first line is no longer a type declaration, but the concatenation of three syntactic elements, including the ":" as one. A third alternative for the first line is to use only an object type, of which abstract is an example. This alternative is not helpful here since there would be no way to reference the unnamed object type. The next three lines are concatenations of three elements, since type is absent, but with a primitive data type stated for each.

Lines 5 and 8 ("push" and "empty") contain object types which are functions. The BNF definition for a function indicates an

optional object type preceeding fun. One has to surmise or be told that this object type represents the result returned by the function. One can possibly by now determine that boolean in line 8 is an object type. For line 5, one has to reexamine the same 5 or 6 definitions to determine that the parenthesized pair is an optionally parenthesized object type consisting of the union of two object types, each of which is represented solely by an identifier since the optional word type has been omitted in both cases. The BNF uses the same convention to indicate that the word type is optional and that not all functions have input parameters and produce results.

The SPECIAL reference manual (Roub77) improves understanding of SPECIAL considerably but the large number of specialized constructs is at least a partial offset. Due to lack of cross-indexing, finding the text in Roub77 which may explain a portion of the BNF definitions can also be a problem. The concept of "type" and "object" need further clarification in SPECIAL, particularly in light of some of the statements made about them, such as (1) there is a distinction made between the objects manipulated by a module and those manipulated by SPECIAL, one being a subset of the other, and (2) it is sufficient to consider a type as a set of possible values. Part of the problem seems to be failure to clearly distinguish between a thing and the name of a thing.

Roub77 confounds the above problems by careless use of plurals. A "designator type" is a particularly important kind of type which is probably equivalent or related to an abstract data type. The principal definition states the "Designator types form a class of objects - designators - that are tokens for objects manipulated by the system being specified." The word "designator" seems to be used primarily to refer to a unique identifier for an object, but it also seems to be used at times to refer to the object itself or to the type of the object. The reference to two sets of objects in the definition is confusing, and it is not clear if the "class" consists of all kinds of designators or a single kind (or type).

SPECIAL and ILPL seem to involve the user in considerable duplication of effort in some cases. Note that in Figure 4 the first six paragraph headings at the top of each of the three columns are identical or nearly so. The example of the Provably Secure Operating System (PSOS) in Neum77 indicates that while the last paragraph in each column (FUNCTIONS, MAPPINGS, and IMPLEMENTATIONS) are frequently rather lengthy, the first six paragraphs can rather easily consume two pages per module in a (MODULE) specification, a MAP, or a PROGRAM MODULE. Much of the material contained in one is contained in the others. A desirable goal would seem to be to enter

the material once and reference applicable portions elsewhere with minimal effort without also referencing that which is improper. More cross-file referencing than seems to be indicated in Boye76 may be desirable.

This condition seems particularly noticeable in the EXTERNALREFS paragraph of a MAP. If module A is to be mapped to modules B, C, and D, then the MAP statement states that the mapping is from "A TO B, C, D". Nevertheless, the EXTERNALREFS paragraph must include declarations for all the primitive objects which appear in the mapping functions, not only primitive objects of B, C, and D but A as well. If two or more different primitive objects with the same name could exist (in different modules), then it would be necessary to identify which object was intended in EXTERNALREFS. However, it seems desirable to avoid such duplicate names and the SRI interface handler requires a unique name for each object in an interface. Therefore, it should be <u>possible</u> to eliminate the EXTERNALREFS paragraph for a MAP, thereby reducing the effort and the volume of product from the methodology. The MAP handler tool would need to access the module source files to do its "internal" consistency check, but this is already done to check external consistency.

In the practical application of the SRI HDM, several features or tendencies can obscure the design and the theoretical basis for the methodology. The first such tendency pertains to the volume of product produced. The HDM de-emphasizes text in its design documentation except for comments. Neum77 describes the design of a Provably Secure Operating System (PSOS) and some applications of it. The design consists of 14 levels. Appendix B contains module specifications (the output from stage 2) for the 11 lower levels (20 modules). The specs are a total of 75 pages long. Although PSOS is probably a relatively large system, 75 pages of material such as that in Figure 8 will probably obscure some of the design details. Although a textual description of such a system would run to many more pages, the reader is much less dependent on each character and word of what he reads for understanding. But the 75 pages is not the end of the products.

PSOS is not carried past stage 2 in Neum77; however, Appendix C takes a small portion of PSOS in a somewhat modified form and proceeds further. The sample problem in Appendix C consists of 3 levels (5 modules). The module specifications (stage 2) total 10 pages. There are 4 MAP's (stage 3) which total 8 pages. One reason there are 4 MAP's is that there are 2 copies of one module in the system which differ from one another slightly but have essentially the same names (more obscuring of the design). Only 2 modules are

implemented (stage 4) but these total 15 pages. Finally the implementation proof (stage 4) is given for one V-function of one short implementation. The proof is 8 pages long although it includes material copied from earlier stages for convenience and is undoubtedly longer than most proofs since it is intended for use as an example. The sheer volume needed to describe the design and implementation of a system has some obscuring effect.

A goal of the SRI HDM is that a module spec (from stage 2) should be independent of other module specifications, to "abolish the intermodule assumptions at the specification level" in the words of Roub77. Roub77 also notes that since this independence is not always possible, the EXTERNALREFS paragraph is provided in a module specification to "describe these intermodule assumptions." This dependence between modules complicates the design process and the design itself and also makes understanding of the design (by a contract monitor, for instance) more difficult.

To determine whether this lack of independence is only a rare problem, one might examine PSOS, the only available, sizable sample of the use of the HDM. As noted above, the lower 11 levels of the design contain 20 modules. (Two or more versions of four of the 20 are included in the 11 levels, but it is not evident from the module specifications what the nature of the differences in the versions is.) Of the 20 modules, 19 of them all make external reference to the 20th. Since this 20th (bottom) module is so basic to the security properties of PSOS, it is not unreasonable to assume that someone wishing to understand any of the other 19 modules should need to understand the 20th module also. However, it was also found that 13 of the 19 make external reference to one or more other modules in addition to the 20th. (EXTERNALREFS are to lower modules; cycles presumably must be avoided.) Thus, the lack of module independence at the specification level is more than just a rare occurrence, in the sample.

A module was selected more-or-less randomly but near the top of the 11 levels. It is part of level 9 (the topmost level of the 11 is level 10). This module specification is four pages long and contains specifications for 9 functions. The specification has EXTERNALREFS to 14 functions, 12 PARAMETERS, and a DESIGNATOR contained in four other modules, one of which is the bottom one. If the names of these external functions and PARAMETERS and the short comments for some of them which are included in the level-9 module are not adequate for understanding, then the lower-level module specifications need to be examined also. If the comments in a lower-level specification are still not adequate, then the specifications for the functions and PARAMETERS need to be examined.

The functions will frequently be defined in terms of other functions, PARAMETERS, etc., of the lower-level module, which may have to be examined. A lower-level function may also be defined in terms of EXTERNALREFS of that module which may involve yet other modules.

TOOLS

The SRI tools deal with modules and INTERFACES and a HIERARCHY of INTERFACES. An INTERFACE is not an abstract machine. It has some of the characteristics of an abstract machine but appears to have been influenced by implementation considerations in that it must include a closed set of modules with regard to the EXTERNALREFS paragraph of each MODULE specification. A HIERARCHY of INTERFACE's may frequently give a distorted view of the corresponding hierarchy of abstract machines. It does not seem clear whether the distortion will hinder use of the HDM or not, but it seems esthetically disturbing at least.

As noted elsewhere, PSOS contains several examples of modules which exist in different versions at different levels (in different INTERFACE's). Differences between these versions do not seem well delineated. It is not even made clear at which stages of the HDM the differences appear. However, Boye76 states that a single interface may contain several instances (versions) of the same module specification (although object names must be different). The effects of such practices need explanation for the methodology and the use of them in a design needs to be clearly defined to avoid confusion.

The specifications for PSOS in Appendix B of Neum77 indicate that either the specifications, the tools or the BNF for SPECIAL have some defects. At least one of the MODULE specifications contains EXTERNALREFS to hidden V-functions in specifications for other MODULE's. At least one instance was also noted of a function with a supposedly invalid section, such as a DERIVATION section for an OV-function, for instance.

DOCUMENTATION AND EXAMPLES

For wider application of software design methodologies, the documentation of them needs considerable improvement. It should be more extensive and of higher quality in the case of some methodologies (with reference to more than just the two examined in this report). Standardization of terminology will help but in the

absence of such standardization, greater use of glossaries and more careful choice of wording is needed.

Many more examples of the use of methodologies are needed. Short, informal, homely examples such as some in Lind76 can help to relate a methodology to past and present experience and practice. Larger, more formal examples are also needed to illustrate the use of a methodology. The problem should be easily explainable and understandable and should be explicitly defined. Sample designs should be without error which implies that they should probably be implemented and tested. Even more useful for one or several such examples would be statistics such as effort required, elapsed time, costs, and numbers and kinds of errors discovered.

REFERENCES

AFR14-2 Air Force Regulation: Acquisition Management: Acquisition and Support Procedures for Computer Resources in Systems, AFR-800-14-II (Volume II), USAF, 26 September 1975, 45 pp.

Boyd76 Boyd, D. L. and G. J. Gustafson, The Design Methodology WELLMADE and its Relationship to the Software Generation Process: An Overview, Report HR-76-131:13-53, Corporate Research Center, Honeywell, Inc., Bloomington, Minn., October 1976. 39 pp.

Boye76 Boyer, R., and O. Roubine, The Hierarchy Specification Environment, an appendage to Roub77, 12 pp.

Dahl72 Dahl, O. J., E. W. Dijkstra, and C. A. R. Hoare, Structured Programming, Academic Press, USA (1972).

Dijk72 Dijkstra, E. W., Notes on Structured Programming, pp. 1-82 of reference Dahl72.

Dijk75 Dijkstra, E. W., Guarded Commands, Non-Determinacy, and Formal Derivation of Programs, CACM, Vol. 18, No. 8, August 1975, pp. 453-457.

Dijk76 Dijkstra, E. W., A Discipline of Programming, Prentice Hall, Englewood Cliffs, NJ, 1976, 217 pp.

Glor77 Glore, J. B., Software Acquisition Management Guidebook: Life Cycle Events (S.A.M. Guidebook #17), ESD-TR-77-22, Electronic Systems Division, AFSC, Hanscom AF Base, Mass., February 1977, 69 pp. (ADA037115).

Haga75 Hagan, S. R., and C. W. Knight, An Air Force Guide for Monitoring and Reporting Software Development Status (Software Acquisition Management Guidebook #4), ESD-TR-75-85, Electronic Systems Division, AFSC, Hanscom AF Base, Mass., September 1975, 98 pp. (ADA016488) NTIS.

Hoar69 Hoare, C. A. R., An Axiomatic Basis for Computer Programming, CACM, Vol. 12, No. 10, October 1969, pp.576-583.

Hoar72b Hoare, C. A. R., Notes on Data Structuring, pp. 83-174 of reference Dahl 72.

Lind76 Linden, T. A., The Use of Abstract Data Types to Simplify Program Modifications, Proceedings of Conference on Data: Abstraction, Definition, and Structure (Salt Lake City, Utah, March 22-24, 1976), ACM SIGPLAN Notices, Vol. 11, 1976 Special Issue, pp. 12-23.

MS483 Military Standard: Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs; MIL-STD-483 (USAF); USAF; 31 December 1970; 119 pp.

MS490 Military Standard: Specification Practices, MIL-STD-490, Dept. of Defense, 30 October 1968, 76 pp.

Neum75 Neumann, P. G., L. Robinson, K. N. Levitt, R. S. Boyer, A. R. Saxena, A Provably Secure Operating System, SRI Project 2581, Stanford Research Institute, Menlo Park, Cal., 13 June 1975, 319 pp.

Neum77 Neumann, P. G., R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson, A Provably Secure Operating System: The System, Its Application, and Proofs, SRI Project 4332, Stanford Research Inst., Menlo Park, Cal., 11 Feb. 1977, 481 pp.

Parn72a Parnas, D. L., A Technique for Software Module Specification with Examples, CACM, Vol. 15, No. 5, May 1972, pp. 330-336.

Parn72b Parnas, D. L., On the Criteria to be Used in Decomposing Systems into Modules, CACM, Vol. 15, No. 12, Dec. 1972, pp. 1053-1058.

Robi77 Robinson, L., and O. Roubine, SPECIAL - A Specification and Assertion Language, Technical Report CSL-46, Stanford Research Inst., Menlo Park, Cal., January 1977, 40 pp.

Roub77 Roubine, O., and L. Robinson, SPECIAL Reference Manual, Technical Report CSG-45, Stanford Research Inst., Menlo Park, Cal., Jan. 1977, 60 pp. (including reference Boye76).

Scho76 Schoeffel, W. L., An Air Force Guide to Software Documentation Requirements (Software Acquisition Management Guidebook #9), ESD-TR-76-159, Electronic Systems Division, AFSC, Hanscom AF Base, Mass., June 1976, 179 pp. (ADA027051) NTIS.